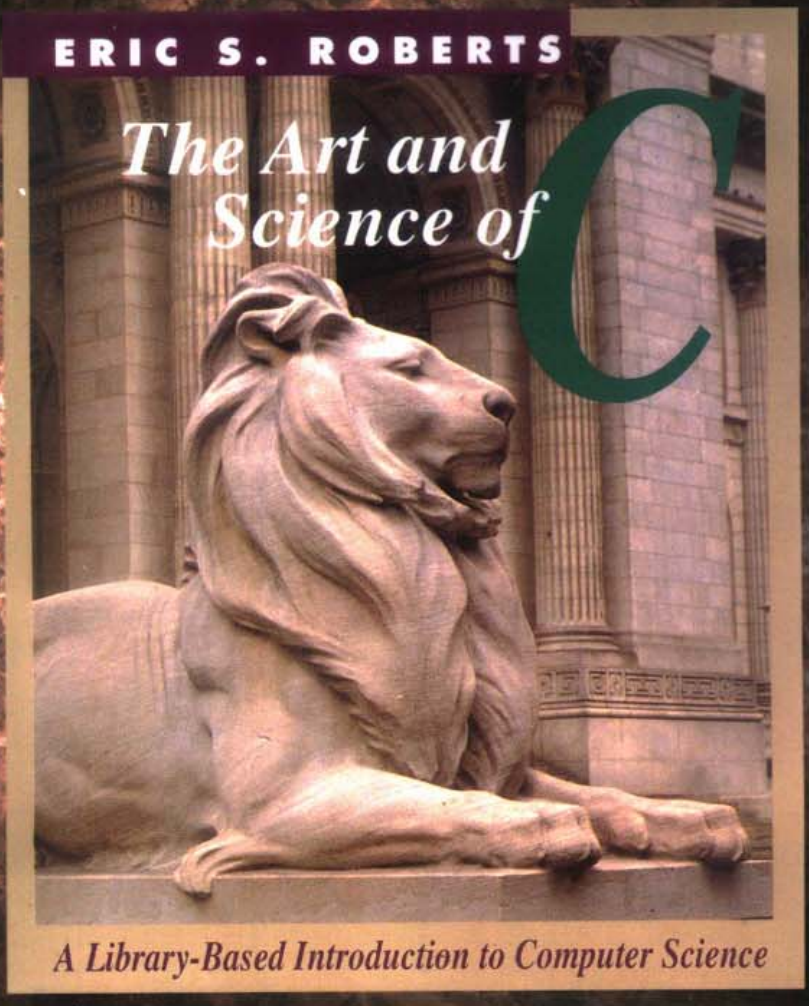
计 算 机 科 学 丛 书

C语言的科学和艺术

(美) Eric S. Roberts 斯坦福大学 著 翁惠玉 张冬茉 杨鑫 蒋文新 译



The Art and Science of C

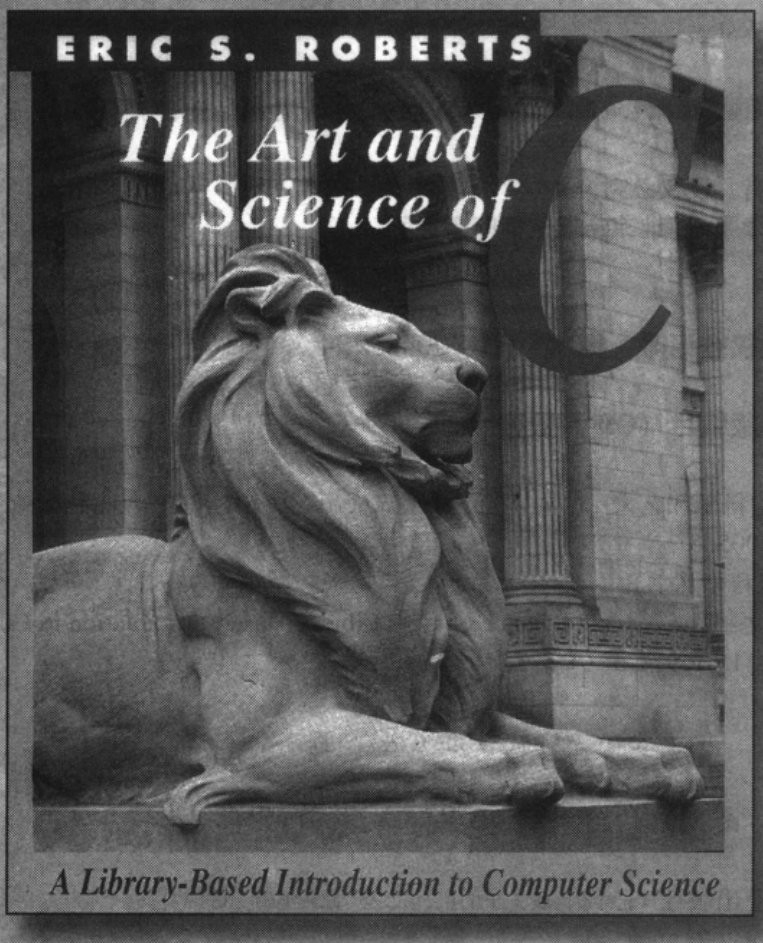
A Library-Based Introduction to Computer Science

机械工业出版社
China Machine Press

计 算 机 科 学 丛 书

C语言的科学和艺术

(美) Eric S. Roberts 斯坦福大学 著　翁惠玉 张冬茉 杨鑫 蒋文新 译



The Art and Science of C
A Library-Based Introduction to Computer Science

机械工业出版社
China Machine Press

本书是计算机科学的经典教材，介绍了计算机科学的基础知识和程序设计的专门知识。本书以介绍ANSI C为主线，不仅涵盖C语言的基本知识，而且介绍了软件工程技术以及如何应用良好的程序设计风格进行开发等内容。本书采用了库函数的方法，强调抽象的原则，详细阐述了库和模块化开发。此外，本书还利用大量实例讲述解决问题的全过程，对开发过程中常见的错误也给出了解决和避免的方法。本书既可作为高等院校计算机科学入门课程及C语言入门课程的教材，也是C语言开发人员的极佳参考书。

Simplified Chinese edition copyright © 2005 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *The Art and Science of C: A Library-Based Introduction to Computer Science* (ISBN: 0-201-54322-2) by Eric S. Roberts, copyright © 1995.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2004-1204

图书在版编目（CIP）数据

C语言的科学和艺术/（美）罗伯茨（Roberts, E. S.）著；翁惠玉等译. —北京：机械工业出版社，2005.3

（计算机科学丛书）

书名原文：The Art and Science of C: A Library-Based Introduction to Computer Science

ISBN 7-111-15971-3

Ⅰ.C… Ⅱ.①罗… ②翁… Ⅲ.C语言—程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2005）第000915号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：朱 劼

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2005年3月第1版第1次印刷

787mm×1092mm 1/16·32.5印张

印数：0 001-4 000册

定价：55.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

专家指导委员会

（按姓氏笔画顺序）

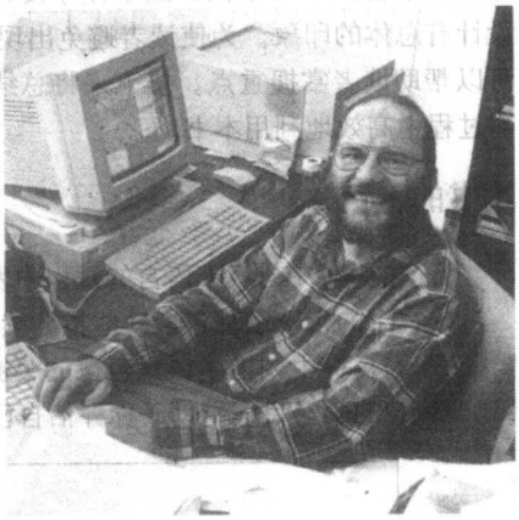
尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

秘　书　组

武卫东	温莉芳	刘　江	杨海玲

前　言

作者简介



Eric S. Roberts第一次教授计算机科学入门课程是在20多年前，当时他还是一名哈佛学子。1980年获得博士学位以来，Roberts先后在哈佛、韦尔斯利、斯坦福大学教授计算机科学课程。在斯坦福大学，Roberts担任计算机科学系副主任，并负责本科生的计算机科学课程。虽然Roberts也教授计算机科学的高级课程并作一些研究工作，但他最大的乐趣还是向初学者展示计算机的强大功能。

在斯坦福大学任教的同时，Roberts自1990年起还担任计算机专业协会主席。这是一个由计算机专业人士组成的公益团体，在全美有22个分会，2000名会员。计算机在多方面影响着我们的社会。学习有关计算机的技术固然重要，但确保计算机服务于社会也很重要。

若读者发现本书中有表述不清楚的地方或错误之处，恳请不吝赐教。请通过Roberts的电子邮箱ericr@aw.com与他联系。

致学生

欢迎你！拿起这本书，你就迈进了计算机科学的世界——这门学科出现在半世纪以前，现在却成为这个时代最具生机和活力的学科之一。

在几十年的发展过程中，计算机几乎使所有领域中看似不可能的事情成为可能。由于计算机可在瞬间将信息传递到任何地方，所以今天的企业家能以空前的规模经营跨国公司。由于计算机可进行必要的、但人工很难完成的计算，科学家才能解决许多问题。电影人利用计算机制作出更具感染力的视觉效果。由于计算机能处理医学中大量的信息处理，因此医生能对患者的病情做出更精确的诊断。

计算机技术正在飞速发展。目前我们已经看到的优势与新的世纪将要经历的发展相比肯定将相形见绌。最近50年，计算机已经对世界产生了深远影响，在新的世纪亦将如此。今日的学生将会是执行这项伟大的工程的中流砥柱。要做到这一点，就必须懂得如何使用计算机。

和其他值得掌握的技能一样，理解计算机的工作原理以及学会怎样控制它们是需要花费时间的。这一切不可能一蹴而就，必须从某个起点开始循序渐进。2500年前，中国的哲学家老子曾说过："千里之行，始于足下"。本书就是一个很好的起点。

然而对很多人来说，万事开头难。许多学生在计算机面前束手无策，认为计算机科学超出了他们的理解范围。可是基本的程序设计并不需要具备高等数学和电子学的知识。在程序

• 小结部分详细地描述了你应该学到的与学习目标部分相关的知识。

小结

第3章从宏观角度展示了程序设计的过程，强调了问题的求解。在此过程中，以非正式的方法介绍了一些控制语句。本章详细地研究了这些控制语句的工作细节。此外，还介绍了一个新的数据类型——布尔型数据。尽管这种数据类型只有TRUE和FALSE两个值，但是能高效地使用它对成功的程序设计十分重要，因此应该多加练习。

本章中还介绍了几个新的运算符，在这里回顾一下已学过的运算符间的优先级关系是有好处的。表4-1总结了这些信息，所有的运算符是按优先级递减的顺序列出的。

表4-1 本章中出现过的运算符的优先级表

运 算 符	结 合 性
一元 ++ -- ! (类型转换)	从右到左
* / %	从左到右
+ -	从左到右
< <= > >=	从左到右
== !=	从左到右
&&	从左到右
\|\|	从左到右
?:	从右到左
op=	从右到左

本章的重要知识点包括：

- 简单语句由表达式跟上分号构成。
- “=”是C语言中用来赋值的运算符。因此，赋值是一个合法的的表达式。它可以以嵌套赋值和多重赋值的形式出现。
- 多个独立的语句可以组合为复合语句，通常称为程序块。
- 控制语句分成两类：条件和迭代。
- genlib库定义了一种bool数据类型，用它来表示布尔型数据。bool型数据只有TRUE和FALSE两个值。
- 使用关系运算符（<、<=、>、>=、==和!=）能够产生布尔值，然后可以使用逻辑运算符（&&、||和!）连接关系表达式。
- 逻辑运算符&&和||是从左到右计算的，一旦能确定整个表达式的值就停止计算。这种行为叫做简化求值。
- 当有些代码仅在特定情况下才执行或程序需要在两条执行路径之间选择其一时可以使用if语句。
- 有些问题有这样的结构：在情况1下完成操作1，情况2下完成操作2，依此类推。此时就可以使用switch语句。
- while语句用来指定在一定条件满足的情况下重复执行某些操作。
- for语句指示一定次数的重复操作，它在每个周期中要更新下标变量的值。

2) 程序设计既是一门科学，也是一门艺术。形成良好的程序设计风格需要掌握很多知识，而不只是记住一组规则。你必须通过实践并阅读其他程序来不断学习。书中在介绍程序设计时的一些特色可以为你提供帮助。

- 本书包括大量的程序实例，这些实例说明了如何用C语句建立一个完整的程序。这些实例也可作为你自己设计程序时的模型。在许多情况下，你可以对书中的程序作简单的修改来解决一个新的程序设计问题。

- 为了帮助你识别和纠正逻辑错误，本书包括了一些说明某些典型错误的程序。为了确保你不会把这些程序作为模型，本书在这些不正确的程序上面叠印了一个苍蝇的图像。



```
/*
 * File: balance2.c
 * ----------------
 * This file contains a buggy second attempt at a program to
 * balance a checkbook.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double entry, balance;

    printf("This program helps you balance your checkbook.\n");
    printf("Enter each check and deposit during the month.\n");
    printf("To indicate a check, use a minus sign.\n");
    printf("Signal the end of the month with a 0 value.\n");
    printf("Enter the initial balance: ");
    balance = GetReal();
    while (TRUE) {
        printf("Enter check (-) or deposit: ");
        entry = GetReal();
        if (entry == 0) break;
        balance += entry;
        if (balance < 0) {
            printf("This check bounces.  $10 fee deducted.\n");
            balance -= 10;
        }
        printf("Current balance = %g\n", balance);
    }
    printf("Final balance = %g\n", balance);
}
```

- 常见错误对所有初级程序员常犯的错误给出了提示，并说明如何避免这些错误。代码中的错误行用苍蝇图案加以突出，旁边给出相应的说明。

在大多数情况下，逻辑表达式不至于复杂到必须用真值表来计算的程度。唯一容易引起混淆的是!或!=和&&或||一起出现的情况。当说起某某情况不为真时（即需要用到!或!=的情况），日常用语和数学逻辑表达有时是相悖的，因此需要格外小心避免犯错。比如，在程序中要表达这样一个意思：*x*不等于2或3。通过这个条件测试语言表述，新程序员很可能会写出以下语句：



```
if (x != 2 || x != 3) . . .          (这是错误的)
```

如果用数学观点来观察这个条件测试，会发现只要x不等于2或只要x不等于3，if测试中的表达式均为TRUE。无论x取什么值，其中一个表达式必定为真，因为如果x为2，则它不可能同时为3，反之亦然。因此上面写出的if测试将永远成功。

常见错误 当&&和||运算符与涉及!和!=运算符的关系测试一起使用时，必须非常小心。自然语言在逻辑上有一些模糊，而程序设计要求非常精确。

- 早日学习C语言为学习C++和面向对象的语言铺平了道路。由于学生在第一年的学习后，就有了较丰富的使用C语言的经验，学校就可以更早开设教学计划中的面向对象的系统设计这门课程。
- C语言中覆盖了一些重要的主题，如模块化开发和抽象数据类型等。这在Pascal语言环境里是很难讲授的。
- 在最近五年中，在工程科学领域的编程中，C语言正在逐步取代Fortran的重要地位。

考虑到近三年的经历，作者确信选用C语言作为计算机科学的入门课程是正确的选择，这种改变将使将来的程序更为强壮。

与此同时，也要意识到在程序设计的早期课程中教授C语言并不是那么容易。C语言是为程序员设计的，并非为作为初学者的学生设计的。其中很多的特性只有在理解了大的概念框架后才有意义，而初学者往往意识不到这一点。从许多方面讲，C语言都是一门复杂的语言，向初学者讲授这门语言时必须很好地控制讲授的进度。

基于库函数的方法

让教师能够控制C语言所固有的复杂性是本教材的中心目标之一。而控制复杂性又是程序员的重要职责。当面对一个很复杂的、不能立即解决的问题时，可将它分解成多个部分，然后单独思考每个部分的解决方案。此外，当某个部分达到了一定的复杂度时，可以将它抽象出来，定义为一个简单的接口。这样，用户就不必仔细理解这个抽象的细节，从而简化了程序的概念结构。

这种方法也适用于程序设计的教学。为使知识更浅显易懂，本书采用了基于库函数的方法，该方法强调抽象的原则。这种方法的特征反映在以下两方面，这也是本书区别于其他入门教材之处：

1) 在开始部分详细地介绍库和模块化开发，这是现代程序设计的基本概念。第二部分集中介绍库、接口、抽象和模块化开发的知识。这样，学生在学习数组之前就可以先学会高效使用这些技术。

2) 本书通过库函数的使用来展示它们的功能。一方面告诉学生库函数可以隐藏复杂性，另一方面向他们说明了这个概念。本书介绍了一些新的库函数以帮助学生写出一些有用的程序，但在他们能消化吸收之前并不讲述库函数所隐藏的细节内容。它们的具体实现放在后面的章节讲述，以使学生能够更深刻地意识到抽象的重要作用。

在1992年，作者曾试图只用ANSI C的库函数来讲授入门课程，但结果并不理想。每个新的主题都要求学生对C语言的其余部分很熟悉，这样便无法开展教学。例如，字符串操作尽管在概念上很简单，但在学生能灵活使用字符串之前，他们必须理解数组、指针和内存分配等机制。最优秀的学生也是到了学期末才理解，而大部分学生则根本没有明白这些概念。自从1993年初引进基于库函数的方法以来，学生理解起来更容易了，对计算机这个学科的了解也更多了。

本书使用的库函数接口和相关实现见附录B。附录B还给出了获取源代码的方法。

讲述顺序

本书的章节顺序与斯坦福大学的入门课程的授课顺序大致相同，只是将第17章的内容放

驱动的教学程序，它综合运用了这些基本的数据结构，是本书中最复杂的一个程序设计示例。

第17章包含了在第一门程序设计课程中经常出现的三个重要主题：递归、抽象数据类型和算法分析。在斯坦福，这些主题在第二门课程中用1/4学期的时间讲述。若决定在第一门程序设计课程中讲述递归，强烈建议早日开始介绍，以便学生有时间去消化吸收。教师可以在第5章后直接讲述递归函数，在第6章后讲述递归算法，也可以在第12章后将递归和算法分析放在一起讲解。

教师资源

教师要获取本书的相关教学资源，请到华章网站www.hzbook.com上申请。

致谢

本书的前身是课程讲义，在此基础上进行了大量改进而形成本书，这很大程度上要归功于使用本书的读者的建议。在这里我要衷心感谢斯坦福大学的讲师们：Jon Becker、Katie Capps、Stephen Clausing、Todd Feldman、Allison Hansen、Margaret Johnson、Jim Kent、Andrew Kosoresow、Mehran Sahami和Julie Zelenski，他们在过去三年的14个班讲授了本书。此外，我还要感谢所有的班导师、助教和教务人员：Felix Baker、John Lilly、Sandy Nguyen、Bryan Rollins以及Scott Wiltamuth，他们提供了必要的后勤支持。

本书中很多好的想法来自于一个计算机入门教学的社团，这是我于1992～1993年倡导成立的。它为解决一些困难问题和对教材进行重要改进提供了一个讨论的平台。在此我想向所有参与其中的人表示感谢：Perry Arnold、Jon Becker、Tom Bogart、Karl Brown、Bryan Busse、Katie Capps、Peter Chang、Scott Cohen、Stacey Doerr、Jeff Forbes、Stephen Freund、Jon Goldberg、Tague Griffith、Matt Harad、Lindsay Lack、Christine Lee、Tricia Lee、John Lilly、Albert Lin、Mara Mather、Hugh McGuire、Jeffrey Oldham、David O'Keefe、Bryan Rollins、Samir Saxena、Nikhyl Singhal、Eric Tucker、Garner Weng、Howard Wong-Toi和John Yong。

同时，我还要感谢所有使用本书并为本书的初稿提出意见的学生们：Adjo Amekudzi、Eric Anderson、Andrew Arnold、Kevin Berk、Kevin Carbajal、Ajit Chaudhari、Alida Cheung、Hye-won Choi、Elisabeth、Christensen、Ralph Davis、Joel Firehammer、Peter Frelinghuysen、Trevor Gattis、Teddy Harris、Heather Heal、Enoch Huang、Ann Lee、Ted Lee、Daphne Lichtensztajn、Lisa Maddock、Allan Marcus、Brad McGoran、Forrest Melton、Adrienne Osborn、Masatoshi Saito、Anne Stern、Ricardo Urena和Nichole Wilson。

在1993～1994年，几位来自其他大学的教师试用了本书的初稿并提出了一些很有价值的建议。我要特别感谢哈佛大学的Margo Seltzer，内华达大学里诺分校的Rob Langsner，马里兰大学（巴尔的摩州）的Richard Chang，拉萨尔大学的Jane Turk和维斯特蒙德学院的Kim Kihlstrom。他们为本书早期阶段的修改提供了大力帮助。

我还要感谢对本书进行审校的人员：

Stephen Allan	犹他州立大学
James Schmolze	塔夫茨大学
Don Goelman	Villanova大学
Michael Skolnick	Rensselaer Polytechnic
Stan Kolasa	Rutgers大学

目　录

出版者的话
专家指导委员会
译者序
前言

第1章　概述 ……1
1.1　计算简史 ……1
1.2　什么是计算机科学 ……4
1.3　计算机硬件简介 ……4
1.3.1　CPU ……4
1.3.2　内存 ……5
1.3.3　辅助存储器 ……5
1.3.4　I/O设备 ……5
1.4　算法 ……6
1.5　程序设计语言和编译 ……7
1.6　编程错误和调试 ……8
1.7　软件维护 ……9
1.8　软件工程的重要性 ……10
1.9　关于C程序设计语言的一些思考 ……10
小结 ……10
复习题 ……11

第一部分　C语言程序设计基础

第2章　通过例子学习 ……14
2.1　“Hello world”程序 ……14
2.1.1　注释 ……15
2.1.2　库包含 ……15
2.1.3　主程序 ……16
2.2　两个数的加法程序 ……17
2.2.1　输入阶段 ……19
2.2.2　计算阶段 ……20
2.2.3　输出阶段 ……20
2.3　有关程序设计过程的观点 ……21
2.4　数据类型 ……22
2.4.1　浮点型数据 ……22
2.4.2　字符串类型的数据 ……23
2.5　表达式 ……25
2.5.1　常量 ……25
2.5.2　变量 ……26
2.5.3　赋值语句 ……27
2.5.4　运算符和操作数 ……29
2.5.5　整型数和浮点型数的结合 ……30
2.5.6　整数除法和求余运算符 ……30
2.5.7　优先级 ……31
2.5.8　优先级法则的应用 ……33
2.5.9　类型转换 ……33
小结 ……35
复习题 ……36
程序设计练习 ……37
第3章　问题求解 ……40
3.1　程序设计习语和范例 ……40
3.1.1　复合赋值习语 ……41
3.1.2　自增和自减运算符 ……42
3.2　解决规模稍大的问题 ……43
3.3　控制语句 ……44
3.3.1　重复N次习语 ……44
3.3.2　迭代和循环 ……45
3.3.3　下标变量 ……45
3.3.4　初始化的重要性 ……47
3.3.5　读入一直到一标志习语 ……47
3.3.6　创造一个更实用的应用程序 ……49
3.3.7　条件执行和if语句 ……51
3.4　一个调试练习 ……53
3.5　格式化输出 ……56
3.5.1　printf的格式码 ……57
3.5.2　控制空格、对齐方式和精度 ……57
3.6　构思一个程序 ……60
3.6.1　程序设计风格 ……60
3.6.2　设计时考虑将来的修改 ……61
3.6.3　#define机制 ……61

6.4.3 估计平方根的泰勒级数展开（可选）……143
6.4.4 泰勒级数近似的实现 ……144
6.4.5 停留在收敛半径之内 ……146
6.5 指定数值类型的大小……147
6.5.1 整数类型 ……148
6.5.2 无符号类型 ……149
6.5.3 浮点类型 ……149
小结 ……149
复习题 ……150
程序设计练习 ……150

第二部分 库和模块化开发

第7章 库和接口：一个简单的图形库 …156
7.1 接口的概念 ……156
7.2 图形库介绍 ……158
7.2.1 graphics.h的基本模型 ……158
7.2.2 graphics.h接口的函数 ……159
7.2.3 软件包初始化 ……163
7.2.4 画直线……163
7.2.5 画圆和弧 ……164
7.2.6 获取有关图形窗口的信息 ……166
7.3 建立自己的工具 ……166
7.3.1 定义DrawBox ……167
7.3.2 定义DrawCenteredCircle ……169
7.3.3 绝对坐标和相对坐标间的切换……169
7.3.4 定义过程的好处 ……170
7.4 解决一个较大的问题……170
7.4.1 使用逐步精化 ……171
7.4.2 实现DrawHouse过程 ……172
7.4.3 寻找共同的模式 ……172
7.4.4 结束分解……173
小结 ……177
复习题 ……178
程序设计练习 ……179
第8章 设计接口：一个随机数库 ……183
8.1 接口设计 ……183
8.1.1 同一主题的重要性 ……184
8.1.2 简单性和信息隐藏的原则 ……184
8.1.3 满足客户的需要 ……185
8.1.4 通用工具的优势 ……185
8.1.5 稳定性的价值 ……186
8.2 用计算机生成随机数……186
8.2.1 确定行为与非确定行为 ……186
8.2.2 随机数和伪随机数 ……187
8.2.3 ANSI C中生成伪随机数 ……187
8.2.4 改变随机数的范围 ……188
8.2.5 将此问题通用化 ……192
8.3 在库中保存工具 ……193
8.3.1 接口的内容 ……193
8.3.2 写random.h接口 ……195
8.3.3 random.c的实现 ……196
8.3.4 构造客户程序 ……197
8.3.5 初始化随机数发生器 ……198
8.4 评价random.h接口的设计 ……200
8.4.1 产生随机实数 ……201
8.4.2 模拟一个概率事件 ……201
8.4.3 在接口中包含头文件 ……202
8.4.4 完成随机数软件包的实现 ……202
8.5 使用随机数软件包 ……202
小结 ……205
复习题 ……206
程序设计练习 ……207
第9章 字符串和字符 ……213
9.1 枚举的原理 ……213
9.1.1 在机器内部表示枚举类型 ……214
9.1.2 将枚举类型表示为整数 ……214
9.1.3 定义新的枚举类型 ……215
9.1.4 枚举类型的操作 ……217
9.1.5 标量类型……217
9.2 字符 ……218
9.2.1 数据类型char ……218
9.2.2 ASCII代码 ……218
9.2.3 字符常量……219
9.2.4 ASCII代码方案的重要特性 ……219
9.2.5 特殊字符……220
9.2.6 字符运算……221
9.2.7 ctype.h接口 ……222

12.2 排序……311
12.2.1 对一个整数数组排序……312
12.2.2 选择排序算法……313
12.2.3 选择排序效率的评估……315
12.2.4 测试程序的运行时间……316
12.2.5 选择排序的算法分析……317
小结……318
复习题……318
程序设计练习……318
第13章 指针……323
13.1 将地址作为数据值……324
13.2 C语言的指针操作……325
13.2.1 在C语言中声明指针变量……325
13.2.2 基本的指针操作……325
13.2.3 特殊指针NULL……327
13.3 通过引用传递参数……327
13.3.1 设计函数SwapInteger……329
13.3.2 用引用调用返回多个结果……330
13.3.3 过度使用引用调用的危险……332
13.4 指针和数组……332
13.4.1 指针运算……333
13.4.2 运算符++和--的新作用……335
13.4.3 指针的自增和自减……336
13.4.4 指针和数组的关系……337
13.5 动态分配……338
13.5.1 Void*类型……339
13.5.2 动态数组……340
13.5.3 查找malloc中的错误……340
13.5.4 释放内存……341
小结……341
复习题……342
程序设计练习……344
第14章 再论字符串……347
14.1 string类型的概念表示……347
14.1.1 字符串作为数组……347
14.1.2 字符串作为指针……349
14.1.3 字符串作为抽象类型……350
14.1.4 字符串参数……350
14.1.5 字符串变量……351
14.1.6 指针和数组变量间的区别……352
14.1.7 决定字符串的表示方法……354
14.2 ANSI字符串库……354
14.2.1 strcpy函数……355
14.2.2 strncpy函数……357
14.2.3 strcat和strncat函数……358
14.2.4 strlen、strcmp和strncmp函数……359
14.2.5 strchr、strrchr和strstr函数……359
14.2.6 ANSI字符串函数的应用……360
14.3 strlib库的实现……360
14.3.1 实现转换函数……360
14.3.2 strlib分配函数的实现……363
小结……364
复习题……365
程序设计练习……365
第15章 文件……368
15.1 文本文件……368
15.2 C语言中文件的使用……369
15.2.1 声明FILE*类型的变量……370
15.2.2 打开文件……370
15.2.3 执行I/O操作……371
15.2.4 关闭文件……371
15.2.5 标准文件……371
15.3 字符I/O……371
15.3.1 文件更新……374
15.3.2 在输入文件中重新读取字符……376
15.4 面向行的I/O……377
15.5 格式化I/O……379
15.5.1 printf的三种形式……380
15.5.2 scanf函数……380
15.5.3 用scanf读入字符串……381
15.5.4 格式化I/O的一个实例……383
15.5.5 使用scanf的局限……385
小结……385
复习题……386
程序设计练习……387
第16章 记录……393
16.1 数据记录的概念……393
16.2 记录在C语言中的使用……394

第1章 概 述

[The Analytical Engine offers] a new, a vast, and a powerful language . . . for the purposes of mankind.

— Augusta Ada Byron, Lady Lovelace, *The Sketch of the Analytical Engine Invented by Charles Babbage*, 1843

学习目标

- 理解硬件和软件的区别。
- 认识问题求解是计算机科学的一个重要组成部分。
- 理解术语算法的含义。
- 理解编译器在高级程序设计语言和低级机器语言之间所起的翻译器的作用。
- 认识程序设计错误的主要类型。
- 认识软件维护的重要性和使用良好的软件工程实践方法的重要性。

1

在计算机技术高速发展的今天，很难想象在1940年计算机还不存在。如今，计算机随处可见，大家都会说，我们生活在计算机时代。

1.1 计算简史

从某种意义上说，计算的使用古已有之。许多早期数学家致力于解决重要的实际问题的计算，如观察兽群中动物的数量，计算小块土地的面积或记录商业交易等。这些活动要求人们开发新的计算技术，有时甚至要求人们发明新机器来帮助完成计算过程。例如，在亚洲使用了几千年的算盘，它是由在杆上滑动的小珠组成，这种计数装置大约在公元前2000年就出现了。

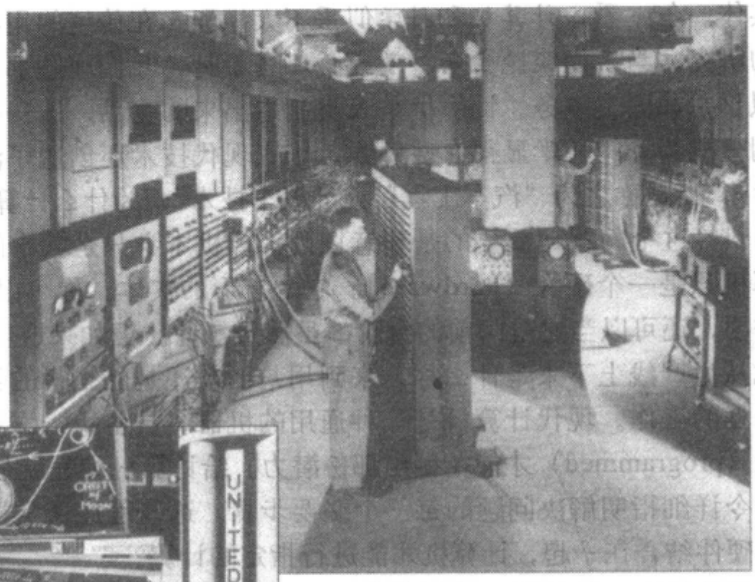
纵观计算的历史，计算的发展相对来说比较缓慢。1623年，德国科学家Wilhelm Schickard发明了第一台机械计算器，这种计算器可以自动执行简单的算术运算。尽管由于三十年战争（1618～1648）的破坏，Schickard的发明已经失传，但法国哲学家Blaise Pascal于17世纪40年代用类似的技术发明了机械加法机，其复制品现存于法国巴黎。1673年，德国数学家Gottfried Leibniz发明了一种相当复杂的装置，除进行加减运算外还可进行乘除运算。所有这些装置都是纯机械的，没有发动机和其他动力源。操作者通过将金属轮转到特定的位置来输入数字，而转动这些金属轮又可带动机器的其他部分，从而改变显示的结果。

在工业革命期间，技术的迅速进步使得用新的方法制造机械的计算机成为可能。当时，人们开始用蒸气机为工厂生产和铁路运输提供动力。在这种情况下，人们开始考虑能否用蒸气机驱动更复杂的计算机器，这种机器可在自己的动力控制下进行复杂的计算。然而在进行这项工作之前，已经有人提出问题并开始寻找答案。在这一领域中，有突出贡献的是英国数学家Charles Babbage，它是计算史上最有趣的名人之一。

Babbage一生中设计了两种不同的计算器，分别称为差分机和分析机，分别代表了当时计算器所取得的巨大进展。可惜的是，他没能完成这些项目。他所设计的用于产生数学函数表的差分机直到30年后的1854年才由一个瑞典发明家实现。分析机是Babbage一生的梦想，但直到1871年Babbage去世时这个梦想也未能实现。尽管如此，他的设计中已经包含了现代计算机的许多基本特点。最重要的是，在Babbage的构思中，分析机是一种通用的机器，可根据设计

2

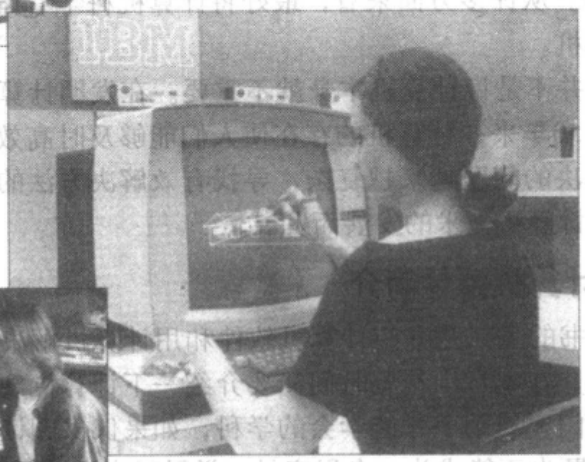
理单元装配在一个单独硅晶片上。这种技术称为**大规模集成**（large-scale integration）。由一个芯片组成的计算机处理器称为**微处理器**（microprocessor），如今已运用于大部分计算机。



通过电子化数据处理，ENIAC使美国人口普查统计数字的制表时间从12年减少为3年。但它的复杂性、高成本和大体积限制了它的吸引力和可用性。



类似于IBM 7090的计算机用穿孔卡片编程，比第一代计算机前进了一步。



集成电路使计算机的体积缩小，但功能增加。随着IBM 360的推出，商用的计算机数量大大增加。



像电话和电视一样，计算机迅速成为一种必不可少的工具，为人们提供从住所连接到世界各地数据库和在线信息中心的通路。

作为现代计算机的鼻祖，第一代和第二代计算机有着重要的历史作用，但现在看来这两种计算机既不有趣也不实用。它们就像计算机科学中的恐龙：庞大、笨拙、智力低下，很快就灭绝了。集成电路的发明者同时也是Intel公司的创始人之一Rober Noyce评论说：“与ENIAC相比，典型的现代计算机芯片比它快了二十倍，有更大的内存，可靠性提高几千倍，消耗的功率仅相当于一只灯泡而非一台机车，体积仅占其三万分之一，成本仅为其万分之一”。计算机技术已臻成熟。

例如，一条指令可能是指示计算机将两个数相加。另一条指令则将一个字符显示在计算机屏幕上。通过执行相应的一组简单指令，计算机可以完成复杂的任务。

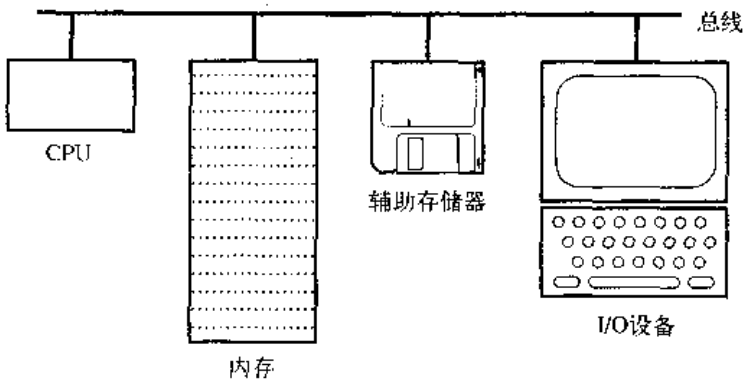


图1-1 典型的计算机组成部分

在现代计算机中，CPU由一个集成电路组成，它是一个小小的硅芯片，上面有无数微型的晶体管。这些晶体管连接在一起可以形成能够执行简单的算术和逻辑操作的电路。

1.3.2 内存

当计算机执行一个程序时，必须有一种方法来存储程序本身和运算所得的数据。总的来讲，计算机硬件中任何能够存储和检索信息的部分都是存储设备。当前运行的程序存放的存储器称为**主存储器**（primary storage），常常称为**内存**（memory）。自从1964年冯·诺依曼首次提出这个构思以来，计算机一直使用同一个存储器来存放组成程序的指令和计算过程中用到的数据。

人们将内存系统设计得十分高效，以使CPU能够以极快的速度获取它需要的信息。在如今的计算机中，内存通常构建在一个特殊的集成电路芯片上，这个集成电路芯片称作RAM，即随机访问存储器，它使程序能够在任何时刻访问任一存储单元。第11章将详细介绍内存系统的结构。

1.3.3 辅助存储器

尽管计算机运行程序时使用内存保存活动数据，但大多数内存有一个缺点，即只有在开机时能存储数据。关机后存储在其中的信息就会丢失。为使数据能永久保存，人们需要使用一种无需电力就可以保存信息的存储设备，这种设备就是**辅助存储器**（secondary storage）。

如今，计算机最常用的辅助存储器是**磁盘**（disk），它由唱片状的圆盘组成，上面涂有磁性物质以记录数据。在现代的个人计算机中，磁盘有两种形式：建造在计算机系统内部的**硬盘**（hard disk）和可移动的**软盘**（floppy disk）。当你编辑程序时，你通常在硬盘上完成你的工作（如果有可用硬盘的话）。当想要将一个程序转移到另一台计算机或出于安全的考虑要进行备份时，通常使用软盘来转移程序。

1.3.4 I/O设备

为了使计算机更加有用，必须有一些方法使其能与外界的用户沟通。计算机的输入通常由从键盘上键入的字符组成。计算机的输出结果常显示在计算机的屏幕上或打印机上。执行输入和输出操作的硬件设备合称**I/O设备**（I/O device），其中I/O表示输入/输出。

不同计算机的I/O设备有很大差别。除了标准的字母键外，不同计算机的键盘有不同的排布，甚至有些重要键的名字也不同。例如，用来表示一行结束的键在有些键盘中表示为Return，

1.5 程序设计语言和编译

用计算机解决问题包括两个概念上不同的步骤。首先，应该构造出一个算法或在解决该问题的已有算法中挑选一个。这个过程称为**算法设计**（algorithmic design）。第二步是用程序设计语言将该算法表达为程序，这个过程称为**编码**（coding）。

刚开始学习程序设计时，将算法翻译成实际的C语言的编码过程会是整个过程中较为困难的阶段。作为一个新程序员，要从简单的问题入手，就像学习任何新技术时所做的那样。简单的问题常有简单的解决方法，它的算法设计阶段不会有太大的挑战性。但由于语言及其规则是全新的，你对之并不熟悉，所以编码常常显得困难和武断。可以肯定的是，当学习了更多的有关程序设计的过程后，编码会很快变得简单起来。然而与此同时，随着要求解决的问题变得更加复杂，算法设计却变得更为困难。

本书在介绍新算法时，都会先用自然语言说明这种算法。尽管这并不像人们想象的那么精确，但在讲英语的人群中，可以用英语来表达解决方案。但如果你要向一个讲俄语的人表达你的算法，那么英语就不再是一个合适的选择。同样，在向计算机表达算法时，使用英语也是不恰当的。尽管几十年来，计算机科学家一直在研究这个问题，但让计算机理解英语、俄语或任何一种其他的人类语言还是现有技术所力不能及的。如果用人类语言表达算法，则计算机将完全不能解释算法。要想让一种算法被计算机接受，就得把它翻译成程序设计语言。 10
世界上有许多种程序设计语言，包括Fortran、BASIC、Pascal、Lisp等许多语言。本书将讲授如何使用C语言，近年来它已成为计算机业界事实上的标准。

上述包括C语言在内的程序设计语言都是被计算机科学家称为**高级语言**（higher-level language）的实例。这些语言的设计不受特性各异的计算机的影响，而是使用通用的算法概念，这种算法概念可运用于任何一个计算机系统。在内部，每个计算机系统都能理解一种低级语言，这种低级语言是它的硬件类型所决定的。例如，虽然Apple公司的Macintosh计算机和IBM的PC机都能执行用C语言这样的高级语言编写出的程序，但它们使用的基本的机器语言是不同的。

为了使用高级语言编写的程序能够在不同的计算机系统上运行，首先必须将程序翻译成运行程序的计算机特有的低级机器语言。例如，若为Macintosh机器写了一个C语言的程序，那么应该运行一个特殊的程序，该程序将C语言转换成Macintosh的专用机器语言。如果在IBM的PC机上运行该程序，则应该使用另一种翻译程序。在高级语言和机器语言之间执行这种翻译任务的程序叫做**编译器**（compiler）。

在大部分计算机系统上，运行程序之前，要先输入程序文本并将其保存在一个**文件**（file）中。文件是存储在计算机辅存里的信息集合的统称。每个文件都必须有一个文件名，通常用句点将文件名分成两部分，如`myprog.c`。建立文件时需要选择一个**根名**（root name），这是文件名中句点之前的那部分名称，它说明文件的内容。句点后的那部分名称作为**扩展名**（extension），说明文件的类型，一些扩展名有预先指定的意义。例如，扩展名.c表示文件中的程序是用C语言编写的。包含程序文本的文件称为**源文件**（source file）。

输入文件或改变文件内容的过程称为**编辑**（editing）文件。各个计算机系统之间的编辑过程差异很大，因此不可能用一种适合于各种类型的硬件的方式来描述编辑过程。当在某个计算机系统上工作时，应当学习如何建立新文件及编辑已存在的文件。这种信息可在计算机的用户手册或所使用的C编译器的说明文档中找到。

它是程序设计过程中重要的一环。

逻辑错误可能会非常难以察觉并且令人沮丧。有时程序员确信程序的算法是正确的，但随后发现它不能正确处理以前忽略了的一些情况；或者也许在程序的某个地方做了一个特殊的假定，但随后却忘记了；又或者犯了一个非常愚蠢的错误。

不用紧张，许多人都和你一样。哪怕最优秀的程序员也会有这样的经历。事实上，所有程序员都会犯逻辑错误。特别是，你还会制造一些逻辑错误。算法是一个棘手的东西，你也许常常觉得自己没能很好地掌握它。

从许多方面来讲，发现自己犯了逻辑错误是通往程序员的道路上的一个重要环节。计算机科学的先驱Maurice Wilkes在描述他在20世纪60年代早期作为一个程序员时的经历中写到：

> 在摩尔学院及其后来的一段时间，人们常常主观地认为保证程序的正确性不会有多大的困难。我清楚地记得就在那一瞬间，我明白我未来生活的很大一部分时间将花在寻找自己程序中的错误上。

优秀程序员的优秀之处并不在于他们能够避免逻辑错误，而是他们能努力将存在于完成的代码中的逻辑错误数量减到最少。当设计完一种算法并将它翻译成合乎语法的程序后，必须知道工作还未做完。几乎可以肯定，程序中的某个地方肯定会存在逻辑错误。作为一个程序员，你的工作就是不断找出并改正这种逻辑错误。应当总是怀疑自己编写的程序，并尽可能彻底地进行测试。

1.7 软件维护

软件开发的一个特殊方面是程序需要维护。事实上，对软件开发的研究表明，大多数程序在发布后，用于维护的费用占到了总费用的80%～90%。然而对于软件来说，很难想象维护到底意味着什么。这种说法听起来很奇怪。考虑一辆轿车或一座桥的情况，只有某些地方损坏时才需要维护，如一些金属部件生锈了，一个机械联动装置因过度使用而磨损了，或一些 13
部件在事故中毁坏了，这时需要进行维护。而这些情况都不会发生在软件上。代码本身不会生锈，一次次重复使用同一个程序不会在任何方面降低其可用性，偶然的错误使用当然会有危险结果，但并不会破坏程序本身，即使破坏了也可以从备份中重建程序。那么在这种情况下维护意味着什么呢？

软件需要维护主要有两个原因。首先，即使经过大量测试，并在相关领域使用多年，源代码中依然可能存在逻辑错误。其次，当出现一些不常见的情况或发生之前未预料到的情况时，之前隐藏的逻辑错误就会使程序运行失败。因此，调试是程序维护的一个基本部分。但它并非最重要的部分。更为重要的部分或许可以称为功能增强（尤其是从其占据程序总体维护费用的百分比的角度来说）。编写程序是为了使用，用程序完成顾客们需要完成的任务通常比其他方法更快更便宜。同时，程序不可能完成顾客想做的每一件事。在一个程序使用了一段时间后，顾客会期望程序能做些别的事情，或者用不同的方法做某些事情，或者用更有用的方式显示数据，或者运行得更快一些，或者有更大的容量，或者有一些更为简单但更有吸引力的特色等等。由于软件具有极大的灵活性，供应商可选择响应这些要求。不管人们是想要纠正逻辑错误还是想要增强功能，在任何情况下都需要有人查看程序，找出需要做什么，做出必要的修改，确保这些修改能正确工作，然后发布新的版本。这个过程既费时又费力，花费昂贵而且易出错。

本章重点介绍程序设计的过程所涉及的内容及其与广义计算机科学的联系。

本章的要点如下：

- 一个计算机系统的物理组件组成了硬件，这些组件是看得见、摸的着的。在硬件能够工作之前必须详细指定一系列指令，即程序，以便告诉硬件应该做什么。这样的程序叫做软件。
- 计算机科学与其说是关于计算机的科学，还不如说是用计算机解决问题的科学。
- 在计算机上解决问题的策略称为算法。成为算法的策略必须清楚明确地定义、有效、有限。
- 程序通常使用高级语言编写，然后编译器将这种高级语言翻译为特定计算机系统所使用的低级计算机语言。
- 要运行一个程序，必须先建立一个包含程序文本的源文件。编译器将源文件翻译成目标文件，它和其他目标文件链接在一起，建立一个可执行程序。
- 程序设计语言有一组语法规则以判断一个程序是否正确构建。编译器按照这些语法规则检查程序，当程序违反这些规则时就报告语法错误。
- 最严重的程序设计错误就是程序语法正确，但显示不正确的结果，甚至根本就不显示结果。由于逻辑上的错误导致程序不能正确地解决问题，这种类型的错误称为逻辑错误。找出并修正逻辑错误的过程称为调试。
- 大部分程序必须周期性地更新以纠正逻辑错误，或应应用的要求修改程序。这个过程称为软件维护，设计程序使之易于维护是软件工程的重要组成部分。
- 本书使用C程序设计语言来说明程序设计过程。尽管人们在用C语言编写程序时很少考虑软件工程的原则，但用C语言可以写出具有良好编程风格的程序。 16

复习题

1. Babbage设计的分析机为计算领域引入了什么新概念？
2. 谁是人们公认的第一名程序员？
3. 冯·诺伊曼体系结构的核心理念是什么？
4. 硬件和软件的区别是什么？
5. 传统科学涉及的是抽象理论或宇宙的本质而不是人造制品，那么，是什么抽象概念形成了计算机科学的核心？
6. 算法必须满足的三条准则是什么？
7. 算法设计和编码的区别是什么？通常情况下哪一个活动更为困难？
8. 术语高级语言的含义是什么？本书使用哪一种高级语言作为基础语言？
9. 为什么在不同厂商生产的计算机上运行程序需要使用不同的编译器？
10. 源文件和目标文件的关系是什么？程序员直接面对的是哪一种文件？
11. 语法错误和逻辑错误的区别是什么？
12. 判断题：优秀的程序员在编程中不会犯错误。
13. 判断题：编写程序最主要的开销来自程序的开发。一旦程序投入使用，程序的耗费就可以忽略不计。
14. 术语软件维护的含义是什么？
15. 为什么在编写程序时应用良好的软件工程原则很重要？ 17

第一部分

C语言程序设计基础

概要

学习C语言或任何程序设计语言在很多方面就像学习使用外语与人沟通一样。我们需要掌握一定的词汇以知道其含义，需要学习语法以便连词成句，还必须了解一些惯用语以理解人们所说的实际意义。因此，第一部分介绍C语言的词汇、语法及惯用语结构。但仅仅掌握这些还不够。就像使用任何语言一样，还必须有话可说。程序设计的精髓在于解决问题。学习如何表达固然重要，但学习如何找出解决问题的方法则是更大的挑战，因而第一部分重点讲解问题求解的策略，以使读者不仅能够像程序员一样编写程序，而且能像程序员一样去思考。

是所有C程序员共有的财富，而你也即将成为其中的一员。程序的文本见图2-1。

```
/*
 * File: hello.c
 * ------------
 * This program prints the message "Hello, world."
 * on the screen.  The program is taken from the
 * classic C reference text "The C Programming
 * Language" by Brian Kernighan and Dennis Ritchie.
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    printf("Hello, world.\n");
}
```

程序注释

库包含

主程序

图2-1 hello.c

该程序本身作为一个文件存储在你所用的计算机系统的永久存储器中。文件名为hello.c，其中.c说明该文件为C语言程序。

如图2-1所示，hello.c程序分为三个部分：*程序注释*、*库包含列表*和*主程序*。尽管其结构非常简单，但它是下面几章将介绍的所有程序的典型，应将其作为C语言程序组织的范例。

2.1.1 注释

hello.c程序的第一部分只是一段英文注释，描述该程序的作用。在C语言中，**注释**(comment)是在"/*"与"*/"之间所有的文字，可以占连续的几行。在hello.c程序中，注释从第一行的/*开始，在第八行的*/后结束。

注释是写给人看的，而不是写给计算机的。他们向其他程序员传递该程序的有关信息。当C语言编译器将程序转换为可由机器执行的形式时，注释被完全忽略。

本书中的每一个程序都以一个专门的、从整体上描述程序操作过程的注释开头，这段注释称为**程序注释**(program comment)。它包括程序文件的名称和一些描述程序操作的信息。本例中的程序注释还说明了程序的来源。注释还可以描述程序中特别复杂的部分，指出可能的使用者，对如何改变程序行为提出一些建议等等。像hello.c这样简单的程序通常不需要很多的注释。但当程序越来越复杂时，读者就会发现给出合适的注释是使程序易读的最好方法之一。

2.1.2 库包含

程序的第二部分由以下两行组成：

```
#include <stdio.h>
#include "genlib.h"
```

这两行说明该程序使用了两个库。**库**(library)是一种工具的集合，这些工具由其他程序员编写，用于执行特定的功能。hello.c程序使用的库是由ANSI C提供的标准输入输出库(stdio)和专为本书设计的通用库(genlib)。本书的每个程序都会用到这两个库，也就是说，每个程序的程序注释结束后都会立即出现这两行。某些程序还会用其他的库，这些程序必须

22
|
23

但printf的作用是什么？和main一样，printf也是个函数，其名称printf代表一组操作。当想要调用这些操作时，只要使用其函数名就可以一起引用这些函数。在程序设计中，通过使用函数名来调用该函数的行为称为**调用**（calling）函数。hello.c程序中的语句

```
printf ("Hello, world.\n");
```

就是对函数printf的调用。

调用一个函数时，常常要提供一些额外的信息。例如，在C语言中，printf是一个在屏幕上显示数据的函数。但应该显示什么数据呢？这些额外信息由一组出现在函数名后的括号内的参数说明。**参数**（argument）是一个函数的调用程序提供给函数的信息。此处，printf被赋予了一个参数，它是一个列于引号中的一串字符，即**字符串**（string）

```
"hello, world.\n
```

该字符串是程序设计语言中数据的第一个例子。C语言中有许多不同类型的数据，应该特别注意如何使用数据的问题。现阶段只需将**数据**（data）看作程序所操作的信息：所有要显示的信息、用户要求的输入、作为计算结果传递的数值或计算过程的中间结果。

在本程序中，main函数中的语句通知库函数printf显示字符串中的所有字符，这些字符作为参数传入printf中。函数printf尽责地逐个显示H、e、l等等，直到屏幕上出现如下所示的完整消息。

```
Hello, world.
```

25

字符串中最后一个字符为特殊字符，称作**换行字符**，用\n表示。当函数printf到达句末的句点时，光标停留于文本的末尾的句点后。若想扩展程序使之显示更多的信息，也许会想在每个新信息开始时另起一行。**换行字符**（newline）使其成为可能，该字符在所有的计算机系统中都进行了定义。当函数printf处理换行字符时，屏幕上的光标移至下一行的起始处，就像敲击了键盘上的Return键一样（有的计算机将该键标为Enter）。在C语言中，程序必须包含换行字符以标记每一行的结尾，否则所有的输出结果将不分行地显示在一起。

2.2 两个数的加法程序

为使读者对C语言程序的工作过程有更好的了解，我们来看一个稍复杂一些的例子。图2-2所示的add2.c程序要求用户输入两个数字，将其相加并显示总和。

add2.c程序中有一些hello.c中没有的概念。首先，add2.c用到了一个新的库——simpio，即简单输入输出库。该库是本书所使用的扩展库，因此在#include行使用引号，如同genlib一样。

```
#include "simpio.h"
```

另一个值得特别注意的新特点出现在main函数的第一行：

```
int n1, n2, total;
```

这一行是*变量声明*的第一个例子。在程序中，**变量**（variable）是一些在编写程序时值未知的数据的存放处。例如，在编写两个数的加法程序时，程序员尚不知道用户要将哪两个数相加。当程序运行时，用户才会输入两个数。为了在程序中引用这些目前尚未确定的数，可创建一

3) **输出**阶段：程序在屏幕上显示计算结果。

2.2.1 输入阶段

在输入阶段，计算机必须要求用户输入两个加数并分别保存在变量n1和n2中。每个加数的输入过程包括两步。首先，程序应在屏幕上显示一个消息以使用户了解程序需要什么，这种类型的消息通常称为**提示消息**（prompt）。和向用户显示的其他消息一样，可使用printf来显示提示消息和其他消息，如下所示：

```
printf ("1st number?");
```

注意，这里在传递给printf的参数字符串后没有换行字符。省略换行字符意味着光标将停在显示的文本后面，紧接在问号之后，如下面的运行示例所示：

```
This program adds two numbers.
1st number? ▮
```

行末的光标提示用户输入一个所需的数值，而提示消息则提示应输入什么样的值。大多数情况下，用于显示输出数据的printf函数调用需要用到换行字符，用于显示输入数据提示消息的printf函数则不需要使用换行字符。

为了读取数据，程序可以这个语句：

```
n1=GetInteger();
```

该语句是一个极为重要的程序结构的实例，该结构称为**赋值语句**。在C语言中，**赋值语句**（assignment statement）将等号右边的数值存储于等号左边的变量中。在本例中，赋值语句的右边是对函数GetInteger的调用，左边是变量n1。

函数GetInteger是simpio库的一部分，用于从用户处读入整数。调用函数GetInteger时，程序等待用户用键盘输入一个完整的数值。用户输入完毕并按下回车键后，该数值作为函数GetInteger的结果返回主程序。在程序设计的术语中，称GetInteger**返回**（return）用户键入的数值。总的来说，这个赋值语句的作用就是调用GetInteger函数，让用户输入数值，并最终将GetInteger返回的数值存储于变量n1中。 28

仔细回顾以下语句的执行是很有好处的：

```
printf ("1st number?");
n1=GetInteger();
```

第一条语句仅在屏幕上显示特定的提示消息，光标停留在该行的末尾。计算机接下来执行语句

```
n1=GetInteger();
```

作为GetInteger函数调用的一部分，程序等待用户键入一个数字值，并将该数字值解释为整型值。例如，输入2时，屏幕的显示如下面的运行示例所示。（为使用户的动作更为清楚，图中用↵符号表明用户按下了回车键以表示输入行的结束，该符号实际上并不出现在屏幕上。本书中用黑体显示用户输入以便与程序产生的输出相区分。）

```
This program adds two numbers.
1st number? 2↵
```

数值2便这样存储于变量n1中了。

其中，下划线部分将由一个十进制整数代替。

要想知道显示的是什么样的整数，printf函数取调用时传来的第二个参数值，此处是变量total。它的值显示在屏幕上，换行符使光标移至下一行。最后一行加上之前已显示于屏幕上的信息就构成了程序add2.c的一个完整的运行示例：

```
This program adds two numbers.
1st number? 2↵
2nd number? 3↵
The total is 5.
```

printf函数可将任何数值作为输出的一部分显示出来。若想将一个整数作为输出的一部分，就应该在printf函数调用的第一个参数中包含格式代码%d。将要显示的实际值可以作为printf的另一个参数，按它们出现的次序排列。例如，若将add2.c的最后一行改为：

```
printf ("%d+%d=%d\n", n1, n2, total);
```

则第一个%d处将显示变量n1的值，第二个%d处将显示变量n2的值，第三个%d处将显示变量total的值。计算机屏幕上最终显示的信息为

```
This program adds two numbers.
1st number? 2↵
2nd number? 3↵
2 + 3 = 5
```

第3章中将详细讨论printf函数。 31

2.3 有关程序设计过程的观点

2.2节详细分析了add2.c程序，讲解了程序中每一个语句的特定功能。要成为一个成功的程序员，就要知道C语言中不同语句的作用和用法。有时，常常要逐句地检查程序，尤其是查找那些使程序不能正常运行的逻辑错误时。但这种详细的观察并不是检查程序的唯一方法。有时回过头从整体上检查程序会更有帮助。

再看一遍add2.c程序中的主程序，试着用一句话归纳它的作用。

```
main()
{
    int n1, n2, total;

    printf("This program adds two numbers.\n");
    printf("1st number? ");
    n1 = GetInteger();
    printf("2nd number? ");
    n2 = GetInteger();
    total = n1 + n2;
    printf("The total is %d.\n", total);
}
```

虽然说printf函数的第一次调用并不是绝对没用，但即使不看前面的说明，不知道不同类型的语句具体是怎么工作的情况下，也很可能看出这个程序的作用。它将两个数相加并显示结果。

也许更为重要的是我们可以对add2.c程序做一些修改来完成一些不同的功能。例如，可

数[⊖]的缩写。若要在程序中存储浮点型数据，就必须声明double型变量，就像在add2.c中必须声明int型变量一样。

要编写一个使用浮点型数值的完整的程序，必须先读取并显示double型数值。与整型数一样，浮点数的读取也是用simpio库中的函数完成的。要读取浮点数，需要调用GetReal函数，它的操作和本章前面2.2.1节中描述的GetInteger函数完全一样，只不过它返回的是一个double型的数值。在屏幕上显示浮点数也需要用到函数printf。但这时不是用%d而是用另一种格式代码。浮点数有多种格式代码，最简单的是用%g，它代表一般的浮点数格式。

图2-3中的编程示例说明了把add2.c程序修改为将两个浮点数相加的程序是很容易的，唯一的不同仅在于改变了变量的类型，将函数GetInteger换成了函数GetReal，将printf行中的%d换成了%g。

```
/*
 * File: add2f.c
 * -------------
 * This program reads in two floating-point numbers, adds them
 * together, and prints their sum.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double n1, n2, total;

    printf("This program adds two floating-point numbers.\n");
    printf("1st number? ");
    n1 = GetReal();
    printf("2nd number? ");
    n2 = GetReal();
    total = n1 + n2;
    printf("The total is %g\n", total);
}
```

图2-3 add2f.c

2.4.2 字符串类型的数据

尽管最初设计计算机是为了解决数值计算问题（因此计算机有时仍称作“number crunchers”），现代计算机处理的大多是文本而非数字。但由于数值计算非常简单，所以本书前几章的程序设计例子将重点介绍如何处理数值数据。实际上，熟练掌握文本数据的处理也很重要。

文本数据的最基本的元素是字符。但将字符组合起来，形成一个连续的单元时才会更有用。在程序设计中，一连串字符称为字符串，它使得在屏幕上显示消息成为可能。从hello.c开始，读者已经在程序示例中见到了一些字符串。认识到字符串也是数据，可以用操作和存储数值数据的方法来处理字符串，这一点很重要。

⊖ 该类型称为双精度浮点是因为它的精度是浮点型float的两倍。C语言开发早期普遍使用float类型，现在则多用double来表示所有的浮点数。

若用作者的名字来运行该程序，将得到如下的运行示例：

```
What is your name? Eric↵
Hello, Eric.
```

字符串对于程序设计来说非常重要，因此本书特地用几章的篇幅来讲述这部分内容。从第9章开始，本书将介绍更多的字符串的知识。

2.5 表达式

当希望程序执行计算时，应写出一个*表达式*以指定必要的操作，其形式类似于数学表达式。例如，要将add2.c程序中变量n1和n2的值相加，则正确的表达式为n1+n2。

C语言中的**表达式**（expression）由*项*和*运算符*组成。**项**（term）表示一个单独的数据值（如前一表达式中的n1和n2）。**运算符**（operator）是一个表示运算的字符或一个字符序列（如+）。在表达式中，项必须是如下选择之一：

- *常量* 程序文本中出现的任何明确的数值都称为**常量**（constant）。像0或3.1415926这样的数就是常量。
- *变量* 变量是数据的存放处，其内容在程序执行期间可以改变。
- *函数调用* 值经常通过调用其他函数来产生，这些函数可以是库函数，这些函数返回数据集给最初的表达式。在程序add2.c中，函数GetInteger用于读取每一个输入值，因此函数调用GetInterger()是一个项。函数调用将在第5章中作进一步讨论。
- *括号中的表达式* 在表达式中可以使用括号来表明操作的先后顺序，就像在数学表达式中使用括号一样。对于编译器来说，在计算开始前，括号中的表达式成为一个必须当作单元来处理的项。

当程序运行时，执行表达式中每一个特定操作的过程称为**求值**（evaluation）。当对表达式进行求值时，每一个运算符都对两边项所表示的数据值进行运算。当所有运算符求值完毕后，剩下的一个数据就是计算结果。例如，给出表达式

```
n1 + n2
```

求值过程包括读取变量n1和n2的值并将二者相加，求值结果就是两者的和。 37

2.5.1 常量

在写数学公式时，常用一些符号表示未知数，用另一些符号表示值已知的常量。例如，已知圆半径（r）求周长（C）的公式为：

$$C=2\pi r$$

为将其转换为程序语句的形式，需要使用变量来保存半径和周长。这些变量随数据的改变而改变，但2和π是值不变的常量。2是一个整型常量，π是一个实数常量，在程序中可用像3.14159这样的浮点数近似表示。由于常量是表达式的重要组成部分，因此给出每一个基本数据类型的常量值就很重要。

1) *整型常量* 要将一个整型常量作为程序的一部分或作为输入数据，只要写出组成该常量的每一个数字即可。若该整数是负数，那么在该数的前面加负号，就像数学中一样。这里

```
anExtremelyLongVariableNameWithADifferentEnding
```

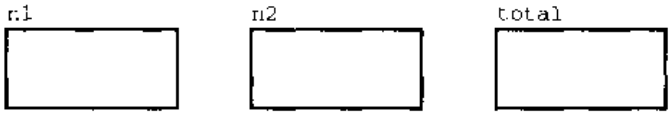
因为二者的前31个字符完全相同。为了防止这种混淆，最好避免使用多于31个字符的变量名。 39

5) 变量名应使读者易于明白其中存储的值是什么。朋友的名字、助词等虽然符合其他几条规则，但对于提高程序的可读性并无帮助。

正如在程序add2.c的讨论中指出的，在程序中使用变量之前必须明确指定每个变量的类型。这个过程称为变量的声明。变量通常在函数的开始部分声明（迄今为止，在本书中声明了变量的函数只有主函数main，但在任何函数中声明变量都是合法的）。声明本身包括变量类型名，后面是一系列作为该类型实例的变量名。例如，add2.c程序中用以下语句：

```
int  n1, n2, total;
```

声明了三个变量。这些变量的名字是n1、n2和total，它们都是int类型。这样，用我们的盒子比喻法，这个声明语句的效果就是创建了以下三个盒子，它们的名字分别为n1、n2和total。

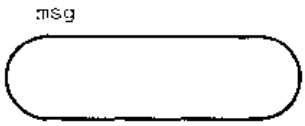


每个变量的初始值都是未定义的，所以在程序开始时不应对三个盒子中的值做出任何假设。变量n1的值可能为73，也可能是其他随机值，只有向它输入值之后才能知道里面的值是什么。

若要声明不同类型的变量，可在函数开头使用其他的声明语句。例如，若要声明string类型的变量msg，可编写以下语句：

```
string msg;
```

用盒子的比喻来说，该声明创建了一个新的名为msg的盒子。



在这个图中，用了另一种形状的盒子以强调变量msg的类型有别于n1、n2和total。例如，变量n1为int型，只能存放整型数据；变量msg为string类型，只能存放字符串数据。试图 40
将错误的数据类型存放于这些变量中，就相当于将方的物体放入圆洞中，编译器会认为这是一种错误。

2.5.3 赋值语句

正如本章前面的add2.c程序所介绍的那样，变量的值是用赋值语句获得的。C语言中的赋值语句形式如下：

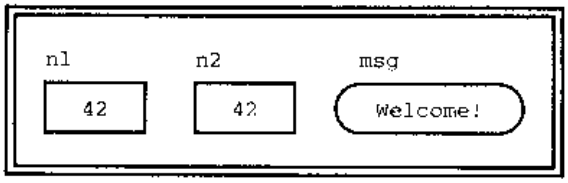
变量=表达式;

在第3章中将会讲到，上面这行语句是*程序设计范例*的一个示例。斜体字表示可以填入任何对应这些类别的内容。在写赋值语句时，等号的左边可以是任何变量名，等号右边可以是任何表达式。范例的剩余部分（本例中为等号和分号）是固定的。要写一个赋值语句，必须以一个变量名开始，然后依次是等号、表达式、分号。

本书在介绍新语句时，将使用下面的*语法框*。语法框简要介绍一种特殊语句类型以便参考。

```
n2 - n1;
```

仅改变变量n2的值，n1的值不变。

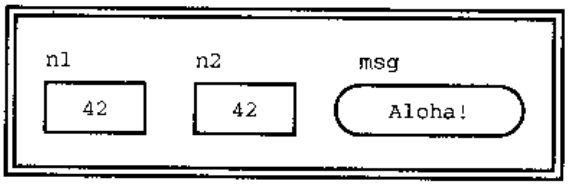


42

为变量赋予新的值将会覆盖变量中原来的值。因此，语句

```
msg = 'Aloha!';
```

将盒子图改变为如下形式：



msg中原先的值将会消失。

2.5.4 运算符和操作数

在一个表达式中，实际的运算是由连接各项的运算符表示的。最简单的运算符是数学表达式中的运算符，其中采用的是算术中的标准运算符。适用于所有数字型数据的算术运算符为：

+ 加法
- 减法（若左边无值则为负号）
* 乘法
/ 除法

每一个运算符左右两边各连接一个小的表达式，形成一个新的表达式。这些次级表达式（或称**子表达式**（subexpression））称为该运算符的**操作数**（operand）。例如，在表达式

```
x + 3
```

中，运算符“+”的操作数为子表达式x和3。操作数常为独立的项，但也可以为更复杂的表达式。例如，在表达式

```
(2 * x) + (3 * y)
```

中，“+”的操作数为子表达式(2 * x)和(3 * y)。

和传统的数学表达式一样，运算符“-”有两种形式。当“-”在两个操作数之间时，例如x-y，表示两个数相减。当其左边没有操作数时，表示负号。因此-x表示x值的相反数。此时运算符“-”称为**一元运算符**（unary operator），因为它只用于一个操作数。其余的运算符（包括表示相减的“-”）称为**二元运算符**（binary operator），因为它们适用于一对操作数。

这些新的运算符使人们可以写出比两个数相加更为有趣和实用的程序。例如，假设要编写一个程序将用英寸表示的长度转化为用厘米表示的长度，只要知道1英寸等于2.54厘米就可以了。可以修改add2f.c程序中的语句，进行适当的调整来写出其余部分。最后结果如图2-5所示。

43

```
9 % 4
```

的值为1，因为9除以4得2，余数为1。以下是%运算符的一些例子：

```
0 % 4 = 0          19 % 4   = 3
1 % 4 = 1          20 % 4   = 0
4 % 4 = 0          2001 % 4 = 1
```

/和%运算符在程序设计过程中使用非常广泛。例如，%运算符常用于检查一个数可否被另一个数整除。若要决定一个整数n能否被3整除，只要看表达式n % 3的结果是否为0即可。

当%两边的两个操作数中有一个带负号或两个均带负号时，C的行为有些混乱。不同的C语言工具处理这种情况的行为是不同的。如果完全依赖于一种行为，那么将程序放在另一台机器上运行时，结果将使人大吃一惊。为确保程序在所有机器上都能以相同的方式运行，应避免使用带负号的%操作数。

2.5.7 优先级

若一个表达式中有多个运算符，则运算的先后顺序就显得很重要了。在C语言中，可用括号括在独立的子表达式外来指定运算顺序。例如，表达式

```
(2 * x) + (3 * y)
```

45

中的括号表明在执行加法运算前，应先执行括号中的乘法运算符。但如果没有括号又会怎样呢？假设表达式为

```
2 * x + 3 * y
```

那么C编译器将如何决定每一个运算符的先后顺序呢？

和大部分的程序设计语言一样，在C语言中，优先级顺序由符合标准数学用法的一系列排序规则决定的，它们被称为**优先级法则**（rules of precedence）。对算术表达式来说，常见的规则为：

1) C编译器首先执行任何一元的负号运算符（即左边无操作数的负号）。

2) C编译器接下来执行乘法运算符（*、/、%）。若两个乘法运算符作用于同一操作数，则优先执行最左边的运算符。

3) 再接下来执行加法运算符（+和-）。同样，若两个加法运算符作用于同一操作数，则优先执行最左边的运算符。

因此，在表达式

```
2 * x + 3 * y
```

中，尽管没有括号，也应先执行乘法运算符。使用括号可使运算顺序更清楚，但本例中无需使用括号，因为想要表达的运算顺序与传统数学中既定的运算顺序相一致。若想改为首先执行加法运算符，则必须使用括号来明确表示这一事实，此时表达式应改为

```
2 * (x + 3) * y
```

优先级法则仅用于两个运算符竞争同一个操作数时。例如，在表达式

```
2 * x + 3 * y
```

中，运算符*和+竞争操作数x。优先级法则规定先执行乘法，因为乘法的优先级高于加法。同样，对于3两边的运算符，用同样的规则可以决定*应该先执行。注意，优先级法则并未明确指定应先执行哪一个乘法。这些子表达式是完全独立的，C编译器可以按任何一种顺序去执

2.5.8 优先级法则的应用

为在实际运用中说明优先级法则，先来看下面的表达式

```
8 * (7 - 6 - 5) % (4 + 3/2) - 1
```

假设自己处在计算机的位置上，将如何对此表达式进行求值呢？

第一步，先对括号内的子表达式求值，可从第一个括号（7－6＋5）开始[⊖]。为计算该表达式的值，用7减去6得1，再加上5得6。因此在对第一个子表达式求值后，式子变为

```
8 * 6 % (4 + 3/2) - 1
```

其中框表示该值是先前对子表达式求值的结果。

在第二个括号括起来的子表达式中，必须先做除法，因为乘法和除法的优先级高于加法。因此，第一步是将3除以2得1（记住，整型数据除法应舍弃余数），然后加上4得5。现在表达式变成以下形式

```
8 * 6 % 5 - 1
```

此时，C语言优先级法则指定应先做乘法和取余数操作，按照这个次序，在作减法之前，先计算6×8＝48，48除以5的余数是3。最后一步是用3减1得2，所以2为整个表达式的值。

2.5.9 类型转换

我们已经学过，在C语言中可将不同数据类型的值结合在一起。此时，C语言用**自动类型转换**（automatic type conversion）来处理这种情况。这个过程隐含在计算过程中，它将一种类型的值转换为另一种兼容的类型。例如，当用数学运算符对一个整型数和一个浮点数运算时，整型数在运算执行之前自动转换为对应的double型数。因此，若写出表达式

```
1 + 2.3
```

48

则在执行加法运算前，计算机内部先将整型数1转换为浮点数1.0。

在C语言中，当对变量赋值时也可进行自动类型转换。因此，若声明变量total为double类型，并将赋值语句写为

```
total = 0;
```

那么作为赋值操作的一部分，要先将整型数0转换为double类型。某些程序设计语言（和一些程序员）坚持将语句写成

```
total = 0.0;
```

其作用是相同的。但0和0.0在数学上的意义是不同的，因此应使用逻辑上与具体应用最适合的形式。0表示数值精确为0，因为整型数是确切值；而在数学或统计学中，0.0通常代表一个接近于0的数，它的值只精确到小数后一位。为避免混淆，本书用整型数表示确切值，即便在浮点数的情况下也是如此。

将一个double型数据赋给一个int型变量时也会激发这种自动转换，在转换过程中会舍掉小数部分。因此，若将变量n声明为int型，则赋值语句

```
n = 1.9999;
```

将会得到令人惊讶的结果，即n＝1。舍去小数部分的操作（此处和整型除法中都会出现）称作**截尾**（truncation）。

⊖ 实际上，C编译器对带括号子表达式的求值顺序是随意的，这取决于对计算机来说哪种顺序更方便，但最终结果是一样的。在编程时应注意避免导致不同求值顺序的情况。

```
quotient = num/den;
```

因为num和den均为整型数。当对两个整型数进行除法运算时，会舍去小数部分。为避免出现这种问题，必须在进行除法运算之前将至少一个值转换为double型。 50

在C语言中，可使用**强制类型转换**（type cast）来明确指出类型转换，强制类型转换是一个一元运算符，由括在括号内的目的数据类型跟上要转换的数值组成。例如，将分母的数据类型转换为double型的语句为：

```
quotient = num/(double) den;
```

既然分母现在为double类型，则按浮点数来计算除法并保留小数部分。同样的，为了转换分子的数据类型，可以将语句编写为：

```
quotient = (double) num/den;
```

这个语句有同样的作用，这是因为强制类型转换的优先级高于除法，这意味着该表达式的计算顺序和以下语句相同：

```
quotient = ((double) num)/den;
```

若强制类型转换运算符的优先级低于除法运算，则C语言将先用一个整数去除另一个整数，舍去小数部分后再将所得的整数转换为double类型，这样就得不到正确的结果。

小结

本章介绍了一些C语言程序的完整例子，使读者对C语言程序的总体结构及工作方式有所了解。主要目标是能用整体的观点看待程序设计过程本身。在本章提供的程序设计示例的基础上，现阶段应能编写包括以下一些操作的简单程序：

- 读取用户提供的数值数据。
- 在屏幕上显示文本和数据。
- 对现有数据进行算术运算得到新的结果。

本章介绍了以下一些程序设计的重要方面：

- 编写质量好的程序都包含注释，注释用自然语言解释程序在做什么。
- 许多程序使用了库提供的工具，这样程序员就不用自己重新编写这些工具了。本书中的所有程序都用到了stdio和genlib这两个库，有的程序还使用了其他库。
- 要使用库，可以在程序开始处加一个#include行以明确指定头文件。如程序示例中所示，像stdio.h这样的系统库的头文件放在尖括号中，个人库及专为本书设计的库（如genlib.h和simpio.h）放在引号中。
- 每个完整的C语言程序都有一个main函数。程序运行时，依次执行main函数中的语句。 51
- 许多程序包括以下三个阶段：输入、计算、输出。
- 接受用户输入的数据时，可以根据数据类型选择使用simpio库中的GetInteger、GetReal和GetLine函数。
- 要在计算机屏幕上显示信息和数据值，可使用stdio库中的printf函数。
- 数据值有许多不同类型，每种类型都由一个值域和一组操作构成。
- 常量用来说明程序中不变的值。
- 变量有三个属性：名字、值和类型。C程序中用到的所有变量必须要先声明，变量声明创建了变量的名字和类型。

点型常量。

a. 42	e. 20	i. 123456789
b. -17	f. 2.0	j. 0.000001
c. 2+3	g. 1,000,000	k. 1.1E+11
d. -2.3	h. 3.1415926	l. 1.1X+11

16. 将下列C语言中的浮点数常量改写成科学计数法表示形式。 53

a. 6.02252×10^{23}	c. 0.00000000529167
b. 29979250000.0	d. 3.1415926535

（每个常数都代表一个化学、物理学或数学中一个重要的近似值：a为阿佛加德罗常数；b为光速（cm/s）；c为用厘米表示的波尔半径；d为数学常数π。用科学计数法表示π并不能带来什么方便，但它是可以用科学计数法表示的，应该知道如何表示。）

17. 指出下列变量名哪一些是C语言中合法的变量名。

a. x	g. total output
b. formula1	h. aReasonablyLongVariableName
c. average_rainfall	i. 12MonthTotal
d. %correct	j. marginal-cost
e. short	k. b4hand
f. tiny	l. _stk_depth

18. 在给一个变量赋值之前，可以对它的值做出什么假设？

19. 说出下列表达式的值和类型：

a. 2 + 3	c. 19.0/5	e. 19 % 5
b. 19/5	d. 3 * 6.0	f. 2 % 7

20. 若变量k已被声明为int型变量，当程序执行赋值语句

```
k = 3.14159;
```

后，k的值是什么？若再执行下面赋值语句后，k的值是什么？

```
k = 2.71828;
```

21. 一元的负号运算符和二元的减号运算符有什么区别？

22. 应用相应的优先级法则计算下列每个表达式的值：

a. 6 + 5/4 - 3

b. 2 + 2 * (2 * 2 - 2) % 2/2

c. 10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1

d. 1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) - 9) - 10

23. 在C语言中如何指定不同数据类型之间的转换？ 54

程序设计练习

1. 将本章中的hello.c程序输入计算机并运行。

2. 下列程序没有为用户提供任何注释和说明，只有几个输入提示消息：

```
#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
```

$$c=\frac{5}{9}(F-32)$$

下面是该程序的运行示例：

```
Program to convert Fahrenheit to Celsius.
Fahrenheit temperature? 212↵
Celsius equivalent: 100
```

若编程时不够小心，则结果总是0。是什么逻辑错误导致这种结果？ 56

8. 在Norton Juster的童话故事*The Phantom Tollbooth*中，数学魔法师给Milo提出下面这个问题：

```
4 + 9-2 * 16 + 1/3 * 6-67 + 8 * 2-3 + 26-1/34 + 3/7 + 2-5
```

根据Milo的计算，该表达式的值总为0，数学魔法师也证实了这一点。但当我们进行计算时，只有从头开始并严格按照从左到右的顺序计算才能使结果为0。若运用C语言的优先级法则执行该表达式将得出什么结果呢？编写一个程序证明之。

9. 编写一个程序，将用千克表示的公制重量换算为用磅和盎司表示的英制重量。它们之间的换算关系为

1千克＝2.2磅　　1磅＝16盎司

10. 编写一个程序来计算四个整数的平均值。

11. 下面是一首儿歌：

As I was going to St.Ives,（在我去St.Ives的路上）

I met a man with seven wives,（我遇到了有七个老婆的男人）

Each wife had seven sacks,（每个老婆有7个麻袋）

Each sack had seven cats,（每个麻袋里有7只猫）

Each cat had seven kits:（每只猫有7只小猫）

Kits, cats, sacks, and wives,（小猫、猫、麻袋和老婆）

How many were going to St.Ives?（有多少人要去St.Ives）

最后一行其实是一个脑筋急转弯问题：只有文中的“我”去St.Ives；其他的都是去向相反的方向。假设要找出有多少小猫、大猫、麻袋和妻子是从St.Ives处过来，请编写一个C程序计算并显示结果。尝试按照问题的结构来编写程序，以便每个读到此程序的人都能理解出它计算的是什么值。 57

使用将会手到擒来。以后每当头脑中产生一个关于解决方案的某方面的大体想法时，将能很自然地把那些想法翻译成恰当的习语，构造出想要的程序。

作为程序设计习语的一个简单例子，考虑第2章中的程序add2.c。程序中的输入部分要求用户输入一个整型值。在任何情况下，add2.c都用下面的语句来完成这个任务：

1) 调用printf来显示提示消息。

2) 调用GetInteger来读入整型值。

60

这些语句构成一个程序设计习语——**读入一个整型数习语**，它的形式如下：

```
printf ("prompt string");
variable=GetInteger();
```

这两行程序是一个程序设计范例（paradigm），即展示特定语句或习语语法结构的C代码段。范例中，斜体部分需用同类成分替换。例如，在此范例中，可以在“*prompt string*”的位置填上任何提示字符串，在“*variable*”部分填上变量名。假如用“1st number”代替*prompt string*，用n1代替*variable*，就得到了add2.c中提示输入第一个数据的语句：

```
printf("1st number?");
n1=GetInteger();
```

用“2nd number?”和n2代替它们，就会得到请求输入第二个数据的语句：

```
printf("2nd number?");
n2=GetInteger();
```

*prompt string*和*variable*能用任何字符串和变量代替。通过替换，可以用同一个基本习语请求输入整型值。

3.1.1 复合赋值习语

有些C语言的习语存在主要是为一些常用操作提供方便的简化形式。其中最重要的是复合赋值操作，它在C程序中是最常见的。

在正式定义复合赋值之前，有必要知道它在什么情况下出现。假设变量balance保存某人银行账户的余额，他想往账户中存一笔钱，数额保存在变量deposit中。新的余额由表达式balance + deposit给出。于是可以写出以下赋值语句：

```
newbalance = balance + deposit;
```

然而在大多数情况下，人们不愿用一个新变量来存储结果。存钱的效果就是改变银行账户中的余额，也就是要改变变量balance中的值，使其加上新存入的数额。我们不使用上面这种把表达式的结果存入一个新的变量（如newbalance）的方法，而是把balance和deposit的值相加，并将结果重新存入到变量balance中，这种方式更可行。赋值语句可修改为：

```
balance = balance + deposit;
```

要了解这个赋值语句的作用，不能将赋值中的“=”看作数学上的等于表达式。在数学上，方程

$$x = x + y$$

61

只有当y取0时可解，除此之外，x不可能等于$x+y$。而赋值语句是一个主动的操作，它将右边表达式的值存入左边的变量中。因此，赋值语句

```
balance = balance + deposit;
```

不是断言balance等于balance+deposit。它是一个命令，使得balance的值改变为它

或

```
y=y-1;
```

++和--运算符只会在C程序中出现[⊖]。然而在第13章中将会介绍比本章所介绍的更有用和更复杂的用法。

3.2 解决规模稍大的问题

程序设计习语和范例只不过是构建程序的砖瓦。当遇到一个问题时，程序员的工作就是将这些砖瓦组织成一个连贯的能解决问题的程序。本章后面的部分将在对第2章的程序add2.c的扩展中介绍几个新的程序设计习语。

图2-2中的程序add2.c读入两个数，将它们相加并打印出它们的和。然而，把两个数相加是一项毫无挑战性的工作，如果所有问题都这么简单，计算机又有什么用武之地呢？计算机的一个最大优势就是能快速处理大量数据。在现实生活中，常见的问题不是如何将两个数相加，而是如何解决更大规模的问题。

假设要修改add2.c使之能累加10个、100个甚至1000个数，那么可以选择同样的解决方案吗？恐怕不行。对10个数来说，利用这个方案或许还能正常工作，但要声明100个变量然后使 63
用单独的语句读入每个变量的值似乎有些不可思议。如果有1000个输入值，那么利用add2.c所用的方案将会导致50多页长的重复的冗长的程序。

现在考虑一下10个数的求和问题。如何处理声明10个变量的问题？解决这样的问题使计算机程序设计有些困难，不过也正是它们使程序设计成为一件有趣的事。静下心来考虑一下这个问题，想象在没有计算机的情况下，别人依次说出10个数：7、4、6……，你将如何计算它们的和？可以将听到的数记下来，最后将它们加起来。这个方案与add2.c所用的方案类似，它的确可行，但却不是明智的和高效的。另一种可选方案是在说出数字的同时将它们加起来：7加4等于11，11加6等于17……。这样不用单独保存每一个数据，只存储当前的和。当听到最后一个数时，也就算出了10个数的和。

这个方案不用保存每个单独的数据，也就回答了如何在不声明10个变量的条件下计算10个数值的和。用这个方案可以写一个add10.c程序，此时只需使用两个变量：一个用于保存每个读入的数据，另一个用于保存当前的和。每当读入一个新的数值时，只需将它加入到保存之前所有数值和的变量中就行了。之后，又可以用该变量保存下一个数值，并按同样的方法处理。

现在可以用新的方案来编写程序了。记住，只需要声明两个变量：一个当前值，一个当前和。在程序开始添加如下的声明：

```
int value, total;
```

当输入每个数值时必须执行下面的步骤：

1) 请求用户输入一个整数值，将它存储在变量value中。

2) 将value加入到保存当前和的变量total中。

如何编写第一步的代码应该很清楚了，这正是前一节中介绍的读入一个整型数习语的典型应用，其代码为：

⊖ ++和--运算符从某种意义上说是C语言的标志性特征。没有其他任何符号能比它们更清楚地表示一个给定的程序是使用C语言而非其他语言编写的。由于这两个运算符在C中使用相当普遍，所以C语言的后继语言（采用了面向对象程序设计技术）被称为C++，因为这个名字意味着“C的后继者”。

这个for语句习语可以作为C语言中编写控制语句的具体范例。在C语言中，控制语句由两个不同的部分构成：

1) *控制行* 控制语句的第一行被称为**控制行**（control line）。它以标识语句类型的关键字开始，常常包含定义整个控制操作的其他信息。在for语句习语中，它的控制行为：

```
for (i=0; i<N; i++)
```

for语句的控制行用来控制大括号中的语句将被执行的次数。

2) *主体* 大括号中的语句构成了该控制语句的**主体**（body）。在for语句中，这些语句将按控制行指定的次数重复执行。为了方便起见，主体内的每一条语句独占一行，并且一般都比控制行多缩进四个空格，这样控制语句的控制范围就一目了然了。

控制行和主体在概念上是相互独立的。一旦写出一个控制语句的控制行，就可以在相应的主体中放入任何需要的语句。因此，for语句能用来重复执行任何操作。例如，执行语句

```
for (i=0; i<2; i++) {
    printf ("a rose is");
}
printf ("a rose.\n");
```

就得到格特鲁·斯坦因的名言

```
a rose is a rose is a rose.
```

for语句可以应用到任何语句集的特性使之成为威力巨大的编程工具。 66

3.3.2 迭代和循环

在程序设计中，重复执行一个操作的过程称为**迭代**（iteration）。迭代是许多问题解决方案的基本组成部分，特别是那些涉及大量数据的问题。一般来说，解决这类问题的程序需要对每个数据执行同样的操作。例如，add10.c对10个输入数据都重复执行读入数据和将数据加入当前和的过程。

程序员常用术语**循环**（loop）来统称在类似for语句的控制语句的执行过程中被重复执行的程序段。循环这个词来源于早期的计算，那时程序是通过穿孔纸带输入计算机的。为了重复一系列的操作，程序员将纸带的首尾相连，使纸带成为一个圈。纸带上的指令传给纸带读入机后，纸带绕一圈回来，指令又被读入，这样便达到了根据需要重复执行的目的。

当一个for循环运行时，计算机顺序执行主体内的每一条指令。当最后一条语句执行完毕时，程序返回到循环开始处并检查是否重复了指定的次数。如果是，程序从整个循环退出，接着执行for语句结尾大括号后的语句。如果还需重复，计算机将重新从循环体的第一条语句起顺序执行每一条语句。循环里所有语句的一次完全执行称为一个**周期**（cycle）。

3.3.3 下标变量

在for语句的控制行

```
for (i = 0; i < N; i++)
```

中，变量i被称为**下标变量**（index variable）。尽管可以使用任何整型变量作为下标变量，但从传统上说通常还是使用i。程序员一看到循环中的变量i，就将它看作跟踪循环的周期数的

3.3.4 初始化的重要性

for循环几乎提供了编写程序add10.c所需的一切，但还有一个重要的细节需要考虑。程序add10.c的核心部分是for循环

```
for (i=0; i<10; i++) {
    printf ("?");
    value = GetInteger();
    total += value;
}
```

它对每一个输入数据重复执行需要的操作。每当读入一个新的数值，程序将它加到变量total中，total用来保存当前所有已经读入的数值的和。

这个方案在程序运行过程中能很好地工作。例如，假设total是123，并且用户输入数值17，程序将简单地将17加入到total保存的值中，并给它一个新值140。但是total第一次的值是多少呢？第一个数值读入变量value后，通过

```
total+=value;
```

加入到total中。

众所周知，这条习语只是令计算机取value的值，将它加入total的当前值，把结果保存在total中。但total的前一次的值是什么？在循环的第一个周期，是不知道这个问题的答案的。在给变量赋值之前，它的值是未定义的。

尽管如此，它的值应该是什么是知道的。为了使程序正确地工作，在for循环的第一个周期之前，total的值必须是0，这样才能使第一个数值加入后得到正确的结果。为了保证total有正确的初始值，需要在循环开始前，明确地将它赋值为0。因此需要在程序的开始部分写下这样的语句：

```
total = 0;
```

用赋值语句来保证一个变量有正确的初始值的过程叫做**初始化**（initialization）。未初始化变量是常见的编程错误。

现在，已经介绍了完成add10.c所需的一切。完整的程序可参见图3-2。

3.3.5 读入一直到一标志习语

程序add10.c是一个说明如何使用for语句的很好的例子，但它现在的这种形式不可能满足很多用户的要求。这个程序能够准确地累加10个数值，若是要用它来累加不同个数的值时，必须要对程序进行修改，尽管这样的修改量很小，但用户真正需要的是能累加任意个数值的更通用的程序。

要想将add10.c修改为能解决累加问题的更通用的程序，有一个只需要对程序进行很小 69
修改的方法。可以在程序的开始部分请求用户输入数据个数，并将之存放在某个变量中，以此来替换for语句控制行中使用的常量10。假设n被声明为一个整型变量，程序的开始几行可以修改为

```
printf ("This program adds a list of numbers.\n");
printf ("How many numbers in the list?");
n = GetInteger();
total = 0;
for (i=0; i<n; i++)  {
...
```

有了这个新的基于标志的循环习语，就可以完成addlist.c了，它的程序参见图3-3。

```
/*
 * File: addlist.c
 * ---------------
 * This program adds a list of numbers.  The end of the
 * input is indicated by entering 0 as a sentinel value.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    int value, total;

    printf("This program adds a list of numbers.\n");
    printf("Signal end of list with a 0.\n");
    total = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetInteger();
        if (value == 0) break;
        total += value;
    }
    printf("The total is %d\n", total);
}
```

图3-3 addlist.c

在第4章中，将会抛开读入-直到-标志习语详细地介绍控制语句。在了解那些细节之前，你就也会发现这个习语是很有用的。对大多数初学者来说，抛开所有疑惑直接使用并不能完全理解的习语的确是有些困难。然而在学习程序设计的过程中，这种情况是很常见的，即使程序设计高手也不例外。实际上，程序设计高手的一大本领就是能不用理解细节而直接使用库函数或现成代码段。随着程序的日益复杂，使用那些只能从大体上理解的工具的能力是一种越来越重要的技能。

3.3.6 创造一个更实用的应用程序

累加一列数据的程序不可能卖出数百万份拷贝，使你成为软件界的百万富翁。而有谁会需要这样的加数程序呢？

当然，很多人会偶尔用到它，但他们很少会以抽象的形式思考这个问题。大多数人都会根据更特定的日常行为来思考，在日常行为中，把一列数相加是很常见的。例如，许多人每个月都会花一定的时间来结算自己的支票簿，这个行为就和加减一列数据类似。要解决这种实用的一些问题，只需要将addlist.c重新包装一下，就可以形成结算支票的程序。

要如何改动原来的程序才能成为支票结算程序呢？这在很大程度上依赖于对它的要求。一开始可以只做如下修改：

1) 修改程序开始部分的注释，以便将来读者能明白程序的作用。

2) 适当更改变量名使之更适合要解决的问题。

3) 为用户提供更明确的提示。

```
Enter the initial balance: 100↵
Enter check (-) or deposit: -50↵
Current balance = 50
Enter check (-) or deposit: -35↵
Current balance = 15
Enter check (-) or deposit: -10↵
Current balance = 5
Enter check (-) or deposit: 50↵
Current balance = 55
Enter check (-) or deposit: -25↵
Current balance = 30
Enter check (-) or deposit: 0↵
Final balance = 30
```

尽管这样的程序并没有达到你期待的通用水平，但至少它的功能对某些人来说是有用的。

3.3.7 条件执行和if语句

假设我们想给这个程序增加一些额外的特性。例如，在上面的例子中需要检测用户何时被银行退过支票。要增加这样的特性，首先要学会编写能做出判断的程序。在编写程序时，经常会遇到这样的情况：有时需要在只有满足某个条件的情况下才执行一些语句，或者根据一些测试的结果在两个可选的过程中选择一个。程序中的这类操作称为**条件执行**（conditional execution）。

C语言中表达条件执行的最简单的方法就是使用`if`语句，可以以如下两种形式使用`if`语句：

```
if (conditional-test) {
    . . . statements executed if the test is true . . .
}
```

或者

```
if (conditional-test) {
    . . . statements executed if the test is true . . .
} else {
    . . . statements executed if the test is false . . .
}
```

当解决方案只有在特定环境下才执行一系列语句时，可以使用`if`语句的第一种形式。在这种情况下，若环境不满足，则那些语句就被跳过。当解决方案需要有两种不同的过程，在满足一定条件的情况下执行其中一个，不满足条件的情况下执行另一个时，可以使用`if`语句的第二种形式。

74

范例中的*conditional-test*（条件测试）部分是一个能提出问题的特殊类型的表达式。第4章将更详细地介绍这类表达式的情况，现在只需要了解由C语言的**关系运算符**（relational operator）组成的非常简单的条件测试就够了。下面列出了C语言中定义的六个关系运算符，同时在后面的括号中给出了与它等价的数学符号。因为某些数学形式（≠、≤和≥）在标准键盘上不存在，所以C语言用两个符号的组合表示这些数学符号。

- `==` 等于(=)
- `!=` 不等于(≠)
- `>` 大于(>)

这个程序与之前版本的唯一不同之处就是下面这条加在while循环末尾的if语句：

```
if (balance<0)  {
   printf ('This check bounces. $10 fee deducted.\n");
   balance --10;
}
```

这条语句的意思是：如果余额小于0，程序将显示一条消息，并在余额中扣除一定的罚金。if语句主体的最后一条语句是

```
balance --10;
```

该语句是

```
balance - balance-10;
```

的缩略形式。

3.4 一个调试练习

75
~
76

支票结算程序中的变化是如此简单，以至于很难有第二种方案。实际上，程序员往往对程序做了一些看似很小的无伤大雅的修改，而不对修改后的程序作彻底的测试。不对代码进行测试是一个非常严重的失误。然而，更严重的失误是没有意识到所有的代码（不管代码有多简单）都需要测试。图3-5中的程序有一个很隐蔽的逻辑错误。假如只对程序进行粗略的测试，程序可以正常工作，这就使得找出逻辑错误更加困难。比如，运行下面这个程序示例，会使程序看起来能够正常工作：

```
This program helps you balance your checkbook.
Enter each check and deposit during the month.
To indicate a check, use a minus sign.
Signal the end of the month with a 0 value.
Enter the initial balance: 100↵
Enter check (-) or deposit: -50↵
Current balance = 50
Enter check (-) or deposit: -60↵
This check bounces. $10 fee deducted.
Current balance = -20
Enter check (-) or deposit: 50↵
Current balance = 30
Enter check (-) or deposit:
```

当用户输入支票金额$60时，balance的值变成负数，程序能正确判断出这个输入值超出了账户中的余额。为了让用户知道这个情况，程序输出一条消息，然后扣除$10罚金。至此，一切都很正常。

如果这样就停止测试，那么将永远不会发现程序中的逻辑错误。让我们再尝试输入另一组数据，只是将上例中最后一次存入的$50改为$10。这一次运行示例的结果如下：

```
This program helps you balance your checkbook.
Enter each check and deposit during the month.
To indicate a check, use a minus sign.
Signal the end of the month with a 0 value.
Enter the initial balance: 100↵
```

```
/*
 * File: balance3.c
 * ----------------
 * This file contains a corrected version of a program to
 * balance a checkbook, including a working bounced-check
 * feature.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double entry, balance;

    printf("This program helps you balance your checkbook.\n");
    printf("Enter each check and deposit during the month.\n");
    printf("To indicate a check, use a minus sign.\n");
    printf("Signal the end of the month with a 0 value.\n");
    printf("Enter the initial balance: ");
    balance = GetReal();
    while (TRUE) {
        printf("Enter check (-) or deposit: ");
        entry = GetReal();
        if (entry == 0) break;
        balance += entry;
        if (balance < 0 && entry < 0) {
            printf("This check bounces.  $10 fee deducted.\n");
            balance -= 10;
        }
        printf("Current balance = %g\n", balance);
    }
    printf("Final balance = %g\n", balance);
}
```

图3-6 balance3.c（正确版本）

就支票结算这个程序来说，应该再多做一些测试。到目前为止，该实例中所用的所有数据都是整数。为了有更好的测试效果，需要使用包含分的数据来运行这个程序。例如，假设账户余额为$100，用户这次写了两张支票，金额分别为$49.95和$19.95。下面的运行示例展示了两次交易后的余额： 79

```
This program helps you balance your checkbook.
Enter each check and deposit during the month.
To indicate a check, use a minus sign.
Signal the end of the month with a 0 value.
Enter the initial balance: 100.00↵
Enter check (-) or deposit: -49.95↵
Current balance = 50.05
Enter check (-) or deposit: -19.95↵
Current balance = 30.1
Enter check (-) or deposit:
```

作为一个程序员，你有责任保证控制字符串中的百分号的数目与控制字符串之外传给printf的表达式数精确地匹配。但是，C编译器无法检查这条规则是否被遵守。假如有一个程序调用printf函数时，它的占位符与值的个数不符，那么继续运行它会产生不可预测的结果。 81

3.5.1 printf的格式码

printf函数的真正威力在于它能以多种格式输出数值（printf最后的字母f表示formatted（格式））。为了能准确地判断出一个值该如何显示，控制字符串中的每个百分号后面都有一个指出输出格式的关键字母。百分号和关键字母的组合称为**格式码**（format code）。在前一节的例子中，格式码是%d，它指定一个十进制整数。除了格式码%d外，程序中还出现过%g和%s，分别表示浮点数和字符串的输出。要注意，每个格式码要求相应的表达式有特定的数据类型。当使用%d时，必须保证提供的表达式值的类型为int。同样，%g需要的是浮点型值的表达式，%s需要的是字符串表达式。然而，C编译器不能检测它们的类型是否匹配，所以在实际使用时要格外小心，确保对参数使用的是适当的格式码。

常见错误 当使用printf函数时，要确保参数的个数与控制串中百分号开头的置换符个数相匹配。此外，还要确保每个参数的类型与对应的格式码是一致的。

下面列出了printf中最常见的格式码，按照应用的数据类型排列。所有可用格式码的完整列表可参阅附录A。

- %d 十进制整型。在%d格式下，数值将以标准十进制计数法的数字字符串形式显示。如果是负数，数值前会加上负号。
- %f 浮点型。在%f格式下，数值作为数字字符串显示，并在适当位置显示小数点。
- %e 指数型。在%e格式下，数值将用标准程序设计语言的科学计数法的形式显示。

 d.ddddde±xx

 这种方法对应于数学中的表示法

 $d.ddddd \times 10^{xx}$

 如果用格式码%E代替%e，那么除了用大写E来表示指数外，输出的结果是一样的。
- %g 普通型。在这种格式下，数值用%f和%e格式中较短的一个显示。如果用格式码%G代替%g，任何以科学计数法输出的数值都会用大写的E。在不能事先确定输出值有多大的情况下，%g可能是最好的输出格式。
- %s 字符串型。在%s格式下，相应的表达式也必须是字符串型，它将字符逐个显示在终端屏幕上。在该字符串中出现的百分号没有什么特殊效果。
- %% 百分号。%%并不真正指明一个格式，它只是说明%本身也是输出的一部分。 82

3.5.2 控制空格、对齐方式和精度

19世纪中叶，当查尔斯·巴贝奇（Charles Babbage）第一次设想研制自动的计算设备时，他的主要动机是让机器来创建数学函数表，以避免手工计算带来的大量错误。今天，计算机仍在创建着大量的表格数据。因此，制表和按格式组织报表仍是实际程序设计中的一个重要方面。

下面显示的这个表最大的特点之一就是数据信息是按列垂直对齐的。作为环境研究的一部分，假设你要创建一个表来展示美国每个州的总面积、森林覆盖面积以及森林覆盖率[⊖]。理想情况下，表的前几行的形式如下所示：

⊖ 数据来源：*The World Almanac and Book of Facts*, New York:Pharos Books, 1992。

里而不是50 751平方英里，它们只能给计算森林覆盖率带来很小的误差。然而，百分率的最后几位对研究来说并不重要。在这个表的第一个版本中，阿拉巴马州的森林覆盖率为66.9%，这个数实际上就足够精确了。通过指定只显示小数点后一位，就能保证表中不包含那些可能并不确切的多余的数字。[⊖]

84

printf函数通过在格式码中加入额外的格式信息来提供控制域宽、对齐方式和输出数据精度的手段。出现在百分号和关键字母之间的看似浮点数的附加信息实际是由下面所介绍的部分组成的，每一个部分都是可选的：

- 负号：它表示域中数据使用左对齐方式。若没有出现负号，则数据将右对齐。
- 数字字段宽度：它指定用于输出字段的最小字符数。假如要显示的值所占用的空间比字段宽度规定的数值小的时候，将在该域中**填充**（padded）额外的空格直到满足一定大小。如果字段宽度前没有负号，将在数据左边加额外的空格以便右对齐。如果出现了负号，那么额外的空格将加在数据右边。注意，字段宽度指定的是最小的宽度。如果数值太大，不能在指定大小的字段中显示，那么只是扩大字段宽度使之能容纳整个数值，尽管这样会打破列的对齐状态。如果没有指定字段宽度，那么数据值将使用自己需要的确切字符位数来显示，不会在两边加空格。
- 小数点及精度参数：这个格式信息的解释依赖于格式码。对%e和%f来说，精度参数指定了小数点后的位数；对%g来说，精度参数说明了最大的有效位数。对%s来说，精度参数说明字符串中最多能显示的字符个数，它避免了过长的字符串对列宽产生的不利影响。如果没有精度参数，那么printf显示整个数据。

前面给出的支票结算程序已经使用了精度参数来改进输出格式。由于支票处理的金额表示为元和分，所以按习惯指定最多显示小数点后的两位数字。用格式说明很容易达到这个目标：

```
%.2f
```

它指定输出的值要保留到小数点后的两位数字。

有了这些格式说明，便能像本节前面的范例那样用printf调用，以整齐的表格方式显示有关森林的数据信息。要让州名出现在14个字符域宽的字符串域的左端，可以使用如下的格式说明

```
%-14s
```

负号说明要左对齐；14说明域宽。如果还想保证新的字符数超过14的州名不会超出列的边界，可以用精度参数指定只显示它的前14个字符。完整的格式说明是这样的：

85

```
%-14.14s
```

使用这个格式，如果printf给定的显示字符串为“District of Columbia”，那么只有前14个字符会出现，如下所示：

```
District of Co
```

⊖ 但是，很多使用表的这种数据的人都认为，所有其中的数据都是准确的，毕竟它们是计算机计算出来的。而事实上，输出数据永远不会比输入数据准确性高，显示过多的数字只能给出精确的假象，对提高统计的准确性并无帮助。为了避免这类假象，确保不要输出超过已知正确范围的多余位数。

条件控制，它们对程序的可读性和结构的清晰性很重要。各种控制语句缩进规则将在第4章中介绍。

- 使用有意义的名字。例如，在支票结算的程序中，变量名balance清楚地告诉读者变量中包含的值是什么。假如只使用一个字母b，可能会使程序更短更容易输入，但对读者来说这个程序的可用性降低了。
- 制定变量的命名规则，使读者能够从名字中了解其功能。在本书中，变量名和数据类型名总是以小写字母开头的，例如n1，total和string等。相比之下，函数名（例如GetInteger）常常以大写字母开头。此外，若函数名是由几个单词组合而成时，就像GetInteger一样，每个单词的首字母都要大写，这样便一目了然。 87
- 在适当的情况下使用标准习语和约定。许多软件公司发布了有关程序设计风格和程序结构的内部规定，其中有些还可能与本教材中的建议相悖。俗话说入乡随俗，如果你使用的产品有内部标准，就应该遵守它，这样才能使其他程序员更容易理解。总的说来，只有在一定范围内对一些基本习惯达成共识后，程序设计才能顺利进行。
- 避免不必要的复杂性。为了程序的可读性牺牲一些程序效率常常是值得的。

我们的宗旨是让程序更容易阅读。为了达到这个目的，最好重新审校自己的程序风格，就像作家校稿一样。在进行程序设计时，最好早一些开始，完成后把它在一边放几天，然后重新把它拿出来。看看对你来说它容易读懂吗？对别人来说将来它容易维护吗？如果发现程序实际上没有意义或不易读懂，就应该花些时间来修改它。

3.6.2 设计时考虑将来的修改

编写程序时，可以采用一些能适应变更的设计以便使将来对程序的修改容易一些。因为程序员知道哪些地方比其他地方更需要修改，所以程序员常常根据经验来判断程序的那些部分应该尽可能设计得灵活些。

回想一下本章前面给出的程序balance.c。在该程序中，程序员最可能会修改的是哪一部分呢？不出意外的话，最可能修改的就是当支票退票时银行所收的罚金。程序员可以轻松地修改这个部分吗？如果现在正在写这个程序，程序员不得不仔细地检查程序，找出$10在程序的那些地方曾经出现过。随后需要做出两处修改。最明显的一处是

```
balance -- 10;
```

然而，更新前一行的printf语句也同样是重要的。

```
printf ("This check bounces. $10 fee deducted.\n");
```

考虑到以后其他人也可能会对这个程序做修改，所以希望只做一处修改，修改效果能传递到程序的所有相关部分。这样，修改程序便更加方便，并且不会因为修改程序的一部分但未修改其他相应部分而破坏程序。 88

3.6.3 #define机制

C语言中进行集中编辑修改的最好的工具就是#define结构。在最简单的情况下，#define有如下的典型形式：

```
#define symbol value
```

在这个范例中，*symbol*是一个名字，它遵循变量的命名规则；*value*是C语言中的一个常量。在#define出现过后，无论*symbol*在程序哪个部分出现，它都将被指定的常量所替代。假如在支票结算程序的开始部分写下这样的语句

```
                    BouncedCheckFee);
                balance -= BouncedCheckFee;
            }
            printf("Current balance = %.2f\n", balance);
        }
        printf("Final balance = %.2f\n", balance);
    }
```

图 3-7（续）

使用#define来设置很可能会修改的值的方法是软件工程的一个重要部分。在后面的内容中，我们将看到更多使用这种技术的例子。

小结

在第2章中，我们学习了如何编写一个简单的程序，让它读入数据，计算结果并产生输出。第3章介绍了控制语句的一些概念，以扩展你的知识。使用控制语句可以让程序解决一些更复杂的问题，如判断一定条件是否满足的测试或需要重复执行的操作。

本章鼓励读者使用控制语句解决一些力所能及的问题。每一条语句都是一个适用于特定情况的工具，之前已经展示了如何在一些简单的习语和范例中使用这些工具。第4章将更详细地介绍控制语句。 89～90

除了熟悉控制语句之外，通过示例程序balance.c的演变，读者可能已经意识到编写程序并不如看起来那么容易。特别是没有经过仔细推敲就对程序作了一些修改，或对程序没有进行彻底的测试，很容易在程序中引入一些不易察觉的逻辑错误，从而使程序得不到想要的结果。虽然从某种程度上说逻辑错误在程序设计过程中是难以避免的，但可以通过良好的程序设计规范来降低问题的严重性并节约大量时间。为了帮助读者形成自己的规范，本章给出了几个指导方针和约定来提高读者的程序设计能力。

本章中介绍的有关程序设计的重点内容如下：

- 程序中常见的操作可以表示为程序设计习语，它提供了能在各种程序设计问题中使用的简单的语句模式。
- C语言定义了几个简单的赋值运算符，使一些常见的操作更加方便。
- 只适用于两三个数据值的解决策略随着问题规模的扩大往往是不可取的。
- for语句能用来以特定次数重复执行一系列语句。
- 当在本章提供的特定的习语中使用while语句时，可以用来反复执行一系列语句直到读入一个设定的标志值为止。
- if语句用来说明一系列语句只有在一定条件满足的情况下才能被执行。而条件常常是用关系运算符比较两个数值来表示。
- 看起来似乎对程序无关紧要的修改可能会引入严重的逻辑错误，对程序应该一直保持怀疑的态度，并对它进行尽可能彻底的测试。
- printf函数提供了大量控制格式化输出的手段
- 编写的程序应该尽可能让其他程序员也容易看懂，编程时，时刻为将来的读者着想是很重要的。

复习题

1. 解释程序设计习语的概念，它在学习程序设计的过程中扮演怎样的角色？ 91

2. 编写程序，分行显示十遍“hello, world.”。

3. 以格特鲁·斯坦因的“a rose is a rose is a rose”例子作为模板，编写一个for循环，使之显示麦克白的悼词中的如下部分：

```
Tomorrow and tomorrow and tomorrow.
```

93

4. 修改程序add10.c，让它累加10个浮点型数据。

5. 编写程序，使用像下面运行示例的格式依次输出1～10的平方值。要求设计时要使上下限值便于修改。

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

6. 传说德国数学家卡尔·弗里德里奇·高斯（1777－1855）在很小的时候就崭露出他过人的数学天赋。在他小学的时候，老师让他计算1～100的和。据说高斯很快就给出了答案5050。编写一个程序解决高斯的老师提出的问题。

7. 编写一个程序，读入用户输入的5个整型值并显示其平均值，如同下面的运行示例：

```
This program averages a list of 5 integers.
? 95↵
? 100↵
? 89↵
? 91↵
? 97↵
The average is 94.4
```

注意，尽管所有的输入值都为整型，但它们的平均值可能会有小数部分。同样，设计时也要注意让使用者能方便地修改输入数据的个数。

8. 修改上题的程序，使程序在一开始时就询问用户要计算均值的数据个数，其运行界面如下图所示： 94

```
This program averages a list of integers.
How many values are there in the list? 5↵
? 95↵
? 100↵
? 89↵
? 91↵
? 97↵
The average is 94.4
```

过15个字符，则只显示其前15个字符。可以假设与会的代表不会超过10 000人，因此票数不会多于4位数字。

这里并不需要编写一个完整的程序来产生整个表，仅仅需要编写printf语句。尽管如此，仍可以试想一下如何测试这个printf语句能否达到预期效果。

13. 在第2章的练习4中编写了一个计算两年复利的程序。重写这个程序使之显示每隔N年的存款余额，其中N由用户输入。

14. 编写一个程序从用户那里读入一系列整型数据直到用户输入作为标志值的0为止。当标志值出现时，要求程序显示之前输入数据的最大值，其运行界面如下所示：

96

```
This program finds the largest integer in a list.
Enter 0 to signal the end of the list.
 ? 17↵
 ? 42↵
 ? 11↵
 ? 19↵
 ? 35↵
 ? 0↵
The largest value is 42
```

在编程之前先考虑一下这个问题，应该用什么解决方案？

如何在一列数据中找出最大值是目前为止本章中最重要的练习。一旦理解了如何解决这个基本问题（不是说如何写出相应的C语句，而是如何设计算法策略），你就可以继续学习一些程序设计的细节内容了。

97

语句。在C语言中，不论其功能如何，所有的语句都遵守以下规则：

> **简单语句规则** 一个简单语句由一个表达式及紧跟其后的分号构成。

100

因此，一个简单语句的范例如下：

```
表达式;
```

在表达式后加上分号就使它成为合法的语句形式。

尽管任何表达式后跟分号都是一个C语言的合法语句，但并不是这样组合肯定会产生有效的语句。要使一个语句有效，它必须有实际的明确的效果。以下语句

```
n1 + n2;
```

由表达式n1+n2紧跟一个分号构成，它是一个合法语句。然而，它是一个无效语句，因为它对计算结果没有进行任何操作。该语句将变量n1和n2的值相加，然后将结果丢弃[⊖]。C语言中典型的简单语句是赋值语句（包括复合形式的赋值和自增/自减操作）和能执行一些有用操作的函数调用语句，如printf。

很明显，根据简单语句规则，如下的程序行

```
printf("Hello, world.\n");
```

是合法语句。根据第2章给出的表达式的定义，函数调用也是合法表达式，因此上面语句的函数调用部分（除去分号那部分）是一个合法表达式。在其后加上分号就成为简单语句。

那么赋值语句呢？如果如下程序行

```
total = 0;
```

符合简单语句规则，那么

```
total = 0
```

本身一定是一个表达式。

在C语言中，赋值所用的等号是一个二元运算符，就像“+”或“/”。运算符“=”带两个操作数，左右两边各一个。目前，可以认为左边的操作数一定是一个变量名，尽管在第11章中会放松此限制。当赋值运算符被执行时，首先计算右边表达式的值，随后将结果存储在赋值号左边的变量中。由于用于赋值的等号是一个运算符，所以事实上

```
total = 0
```

的确是一个表达式，而

```
total = 0;
```

就是一个简单语句。

101

4.1.1 赋值的嵌套

前面对赋值的描述大家应该并不陌生，因为它和之前在第2章的2.5.3节中给出的不太正式的定义是等效的。现在有一个有趣的想法：如果一个赋值语句是一个表达式，那么表达式本身应该有值。而且，如果一个赋值表达式产生一个值，那么也一定能将这个赋值表达式嵌入到一些更复杂的表达式中去。

当赋值作为更大的表达式的一部分时，该赋值子表达式的值即为它用来赋值的值。例如，

⊖ 某些C语言编译器对这类无效语句会给出一个警告。

个独立的语句，它们都是主程序体的一部分。

为了说明一连串的语句是一个连贯的单元，可以把这些语句组装成一个**程序块**（block），用大括号把它们括起来如下所示：

```
{
    statement1
    statement2
    statement3
    . . .
    statementn
}
```

在前面章节的一些程序实例中曾经出现过程序块。每个主程序本身就是一个程序块，第3章的程序中的控制语句体也是程序块。 103

在3.6.1节中曾经讨论过，一个程序块中的语句常常根据上下文的关系有不同程度的缩进。编译器会忽略这些缩进，但其直观的效果对其他读者却是很有帮助的，因为你的程序的结构会因为这些缩进而跃然纸上。经验表明，每个新层次开始时用三到四个空格缩进会使程序的结构非常清晰，本书中每个新层次开始时将采用四个空格。适当的缩进对良好的程序设计来说是很重要的，必须要努力使自己的程序有一致的缩进风格。

程序块中唯一容易使初学者混淆的问题就是分号所扮演的角色。在C语言中，分号是简单语句语法的一部分，它的作用主要是语句*终结符*而非分隔符。虽然这条规则很简单，但对曾经使用过Pascal语言的人来说常常会有问题，因为Pascal的语法规则略有不同。具体来说，主要有以下两点：

1) 在C语言程序块的最后一个简单语句后面总会有分号。而在Pascal中常常不会出现分号，尽管大多数编译器允许将分号作为可选项。

2) 在C语言程序块的结束大括号后不会有分号。在Pascal的关键词END后，根据上下文可以有分号，也可以没有分号。

在C语言中使用分号的习惯有利于程序的维护，并且当你习惯后不会造成任何问题。

C编译器在遇到程序块时，会将整个程序块看作是一个简单语句。不管何时在习语或程序范例中看到“statement”（语句）符号，都可以用简单语句或程序块替代它。为了强调就编译器而言它们是语句，程序块有时也被称为**复合语句**（compound statement）。

4.2 控制语句

如果没有任何转向指示，C程序的语句将按照出现的次序顺序执行一次。然而，对于大多数应用来说，只有自上而下的程序顺序是不够的。现实问题的解决策略往往要求能反复执行一系列步骤或在多个动作中选择一个。能够影响其他语句的执行方式的语句叫做*控制语句*。

C语言中的控制语句分成两大类：

1) *条件*：在解决问题时，常常需要根据一些条件测试的结果，在两个或多个独立的程序执行路径中做出选择。比如，有的程序会要求当某个值为负时执行一系列操作，否则执行另一些操作。需要做出决定的那些控制语句称为**条件语句**。在C语言中，有两种条件语句的形式，即第3章介绍的`if`语句，以及本章后面将介绍的`switch`语句。 104

2) *迭代*：有时要处理的问题涉及很多数据项，则程序通常需要将某一操作重复执行若干次，或在某个条件成立的情况下反复执行某个操作。在程序设计中，这样的反复叫**迭代**。在C

```
if (x=0) . . .                    (这是错误的)
```

的结果很容易引起混淆。这条语句不会检查x的值是否为0，而是将x赋值为0，然后C语言把它理解（具体原因在这里不便说明）为测试结果为FALSE。正确的判断变量x的值是否为0的写法如下：

```
if (x==0) . . .
```

要小心避免犯这样的错误。在输入程序时稍加注意将节省大量的调试时间。

关系运算符只能用来比较**原子数据**（atomic data）值——不能再分解成更小的部分的数据。例如整型、浮点型、布尔型以及第9章中将会介绍的字符型可以组成原子数据，因为它们不能分解为更小的数据。而像字符串型（string）就不是原子数据，因为它由独立的字符组成。因 106
此可以用关系运算符比较int、double或bool型数据，但不能用这些运算符比较string型数据。在第5章中将介绍比较字符串的机制。

4.3.2 逻辑运算符

除了关系运算符外（它取任何类型的原子值并产生布尔型的结果），C语言还定义了三个运算符，它们的操作数为布尔型的值，把它们组合起来形成另一个布尔值：

- ! 逻辑非（如果后面的操作数为FALSE，则值为TRUE）。
- && 逻辑与（如果两个操作数均为TRUE，则值为TRUE）。
- || 逻辑或（如果其中一个操作数为TRUE，则值为TRUE）。

这些运算符叫做**逻辑运算符**（logical operator）。上面的运算符是按优先级递减的顺序排列的。

运算符!、&&、||可以简称为与、或、非。但要记住，有时简称并不一定能准确地表达逻辑意义。为了避免这种不准确，用更形式化、更数学化的方法来看待这些运算符是更有用的。逻辑学家用**真值表**（truth table）来定义这些运算符，它清楚地展现出布尔表达式的值如何随着操作数的值的变化而变化。例如，给定布尔值p和q，&&运算符的真值表为：

p	q	p&&q
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

该表最后一列显示，根据第一、二列布尔变量p和q的值计算出的布尔表达式p&&q的值。因此，真值表的第一行表明当p的值为FALSE，q的值为FALSE时，表达式p&&q的值为FALSE。

||操作的真值表为：

p	q	p\|\|q
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

要注意，不能把||运算符按“或”的字面意思理解为其中一个或另一个成立，而应记住它的数学意义：至少有一个成立。 107

!操作有以下简单的真值表：

常见错误 测试一个数是否在某一特定范围内，只用关系运算符把它们组合起来（就像数学中常用的方法那样）是不够的。条件的两个部分必须明确地用&&连起来，如(0 < x) && (x < 10)。

4.3.3 简化求值

C语言对||和&&运算符的解释与其他很多程序设计语言对II和&&的解释不太一样。例如在Pascal中，这些运算符（在Pascal中记作AND和OR）需要计算出条件两边的表达式的结果，尽管有时结果在计算到一半时就能确定。C语言的设计者们采用了一个不同的，对程序员来说更方便的方法。

当C程序在计算如下形式的表达式

exp1 && *exp2*

或

exp1 || *exp2*

时，子表达式总是从左到右各自被计算。一旦能确定结果就中止计算。例如，若&&表达式中的*exp1*为FALSE，则不需要计算*exp2*，因为结果能确定为FALSE。同样，在||表达式的例子中，如果第一个操作数值为TRUE，就不需要计算第二个操作数。像这种只要结果确定就停止计算的方法叫做**简化求值**（short-circuit evaluation）。

简化求值最主要的好处在于第一个条件能控制第二个条件的执行。在很多情况下，复合条件的第二部分只有在第一部分满足某个条件时才有意义。比如，要表达以下两个条件。第一，整型变量x的值非零；第二，x能整除y。由于表达式y % x只有在x不为0时才计算，所以用C语言可表达这个条件测试为：

```
(x != 0) && (y % x == 0)
```

相应的表达式在Pascal中将得不到预期的结果，因为无论何时它都要求计算出&&的两个操作数。如果x为0，尽管看起来对x有非零测试，Pascal程序还是会因为除零错误而中止。该例中 109

```
(x!=0)
```

这样的为了防止随后部分出现计算错误的条件测试叫做**监护条件**（guard）。

4.3.4 标志

bool型变量在程序设计中十分重要，它还有个特殊的名字：**标志**（flag）。例如按如下方式声明一个布尔型变量：

```
bool done;
```

变量done便成了一个标志，可以用它来记录程序中是否完成了一定的操作。可以像对其他任何变量一样给标志赋值。例如，可以写：

```
done=TRUE;
```

或

```
done=FALSE;
```

更重要的是，可以将任何值为布尔型的表达式赋值给一个布尔变量。比如，某个程序要求只要变量itemsRemaining的值为0，立即结束一些特定的操作，那么可以用如下方法给done赋予合适的值：

```
done = (itemsRemaining == 0);
```

• 该年份能被4整除但不能被100整除。

• 年份能被400整除。

假设表示年份的变量为y，以下布尔表达式将给出正确结果：

```
((y % 4 == 0) && (y % 100 != 0))  ||  ( y % 400 == 0)
```

如果考虑C语言的优先级规则，这个表达式中的括号实际上都不需要，使用括号只是为了使较长的布尔表达式更容易阅读。如果计算这个表达式的结果，并将它存储在一个叫isLeapYear的标志中，就可以在程序的其他部分通过测试标志变量得知闰年的条件是否成立。图4-1给出了计算闰年的程序。 111

```
main()
{
    int year;
    bool isLeapYear;

    printf("Program to determine whether a year is a leap year.\n");
    printf("What year? ");
    year = GetInteger();
    isLeapYear = ((year % 4 == 0) && (year % 100 != 0))
                 || (year % 400 == 0);
    if (isLeapYear) {
        printf("%d is a leap year.\n", year);
    } else {
        printf("%d is not a leap year.\n", year);
    }
}
```

图4-1 leapyear.c的主程序

4.4 if语句

C语言中表示条件执行的最简单办法就是使用if语句，它有以下两种形式：

```
If (condition) statement
If (condition) statement else statement
```

这个范例中的*condition*部分是一个布尔表达式，语句部分可以是简单语句，也可以是程序块。

当解决方案需要在满足特定条件的情况下执行一系列语句时，可以用if语句的第一种形式。如果条件不满足，构成if语句主体的那些语句将被跳过。例如，第3章balance.c的例子中的语句：

```
if (entry<0 && balance<0) {
    printf('This check bounces. $10 fee deducted.\n');
    balance-=10;
}
```

就属于这种类型：支票可能被退也可能没有被退，而程序只在支票被退的情况下才执行这些动作。

当程序必须根据测试的结果在两组动作中选择其一时，可以使用if语句的第二种形式。图4-2的程序oddeven.c说明了这种形式。该程序读入一个数，将它按奇偶分类。决定结果的条件表达式为：

```
n%2== =0
```

112

在这种情况下，如使用大括号并将if语句从一行扩展为三行，将使程序无谓地变长且不易阅读。到目前为止，这种类型的唯一例子出现在第3章的3.3.5节中：

```
if (value == Sentinel) break;
```

4.4.2 多行if语句

当if语句的主体有多条语句或有一条较长的语句使之不能与if放在一行时，这些语句将被划分为一个程序块，如语法框所示。

语法：多行if语句

```
if (condition) {
    statements;
}
```

其中：*condition*是将要测试的布尔值。

*statements*是当条件为TRUE时将要执行的一个语句块。

在这种情况下，如果条件为TRUE，将执行这些语句。如果条件为FALSE，程序不做任何动作，继续执行if语句的下一条语句。

4.4.3 if/else语句

为了避免悬空else问题，带有else子句的if语句主体将用大括号划分为两个程序块，如语法框所示。从技术上讲，只有当条件语句的主体包含不止一条语句时，才有必要在程序块前后加大括号。然而从系统地使用括号的角度上讲，加上括号会使程序产生混淆的情况大大减少，更有利于程序的维护。

语法：if/else语句

```
if (condition) {
    statements_T
} else {
    statements_F
}
```

其中：*condition*是将要测试的布尔值。

$statements_T$是当条件为TRUE时将要执行的一个程序块。

$statements_F$是当条件为FALSE时将要执行的一个程序块。

114

4.4.4 级联if语句

下面的语法框展示了当实际有两种以上情况时，if语句的一个十分重要而特殊的用法：

语法：级联if语句

```
if (condition_1) {
    statements_1
} else if (condition_2) {
    statements_2
} else if (condition_3) {                    (可以出现任意多次)
    statements_3
```

其中存储在变量value中的值即为整个?:表达式的值。

例如，可以用?:运算符将x和y中值较大的一个赋值给max，程序如下：

```
max = (x > y) ? x : y;
```

使用?:运算符的一种最常见的情况是调用printf进行输出时，输出结果可能因为某个条件而略有不同。比如编写一个程序统计某物品的数量，统计结束后将物品数量存储在变量nItems中。怎样把结果报告给用户呢？最直接的方法就是像下面这样调用printf函数：

```
printf ("%d items found.\n", nItems);
```

但假如遇到一个语言学者，当nItems的值碰巧为1时，看到这个输出可能会让他很不满意：

```
1 items found
```

因为其中复数形式的items应该是单数的。然而，可以通过在if语句中包含两个printf语句来更正这个英语语法错误：

```
if (nItems>1) {
    printf ("%d items found.\n", nItems);
} else {
    printf ("%d item found.\n", nItems);
}
```

116

遗憾的是，解决这样一个简单的问题却用了五行语句。其实，这里就可以用?:运算符来解决：

```
printf ("%d item%s found.\n", nItems, (nItems > 1) ? "s" : "");
```

输出时，如果nItems的值大于1，则在字符串item后加上一个字符串s，反之就加上一个空字符串。

另一个例子就是用?:运算符以自然语言方式输出布尔变量的值。值得注意的是，bool类型不是标准C语言的一部分，因此没有内建的输出布尔型值的机制。尽管如此，还是可以使用printf和?:来显示布尔型变量errorFlag的值：

```
printf ("errorFlag = %s\n", (errorFlag) ? "TRUE" : "FALSE");
```

在C语言中，可能会过度使用?:运算符，如果一个程序中重要的判断结构隐藏在这种嵌套的?:运算符中，后面的代码很可能就会遗漏这个判断。另一方面，如果可以使用?:代替复杂的if语句来处理一些小的细节，那它将大大简化程序的结构。

4.5 switch语句

当一个程序逻辑上要求根据特定条件做出真假判断并执行相应动作时，if语句是理想的解决方案。然而还有一些程序需要更复杂的判断结构，它们有两个以上的可选项，这些选项被划分为一系列互相排斥的情况。比如，在一种情况下，程序应该执行x；在另一种情况下，应该执行y；在第三种情况下，应该执行z，等等。在许多应用中，最适合这种情况的就是switch语句，它的语法如下框所示：

语法：switch语句

```
switch (e) {
```

时，将继续执行第二个子句。从概念的角度看，最好还是不要将这种结构看作是一个代表两种可能的case子句。

在switch语句中，default子句是可有可无的。假如没有default子句，当任何case子句都未匹配成功时，程序不做任何动作，继续执行switch结构之后的语句。为了避免程序忽略一些意外情况，在每个switch语句中都使用default子句是一种良好的程序设计习惯，除非确定已经在case语句中列举了所有的可能情况。 118

由于switch语句通常可能会很长，如果case子句本身较短，那么程序比较容易阅读。如果有足够的空间将case关键字、子句的语句和break语句放在同一行会更好。图4-4中的程序cardrank.c就是这种switch语句风格的一个例子，它是switch语句在一个玩牌程序中的应用。在这个游戏中，每一组牌用数字1～13代表，当要显示这些牌时就遇到问题了，2～10可以直接显示数字，但1、11、12和13不能直接显示数字，而应该用Ace、Jack、Queen和King来表示。程序cardrank.c用switch语句为每张牌显示适当的符号。

```
main()
{
    int n;

    printf("What is the rank of the card (1-13)? ");
    n = GetInteger();
    switch (n) {
      case  1: printf("Ace\n"); break;
      case 11: printf("Jack\n"); break;
      case 12: printf("Queen\n"); break;
      case 13: printf("King\n"); break;
      default: printf("%d\n", n); break;
    }
}
```

图4-4 cardrank.c的主程序

switch语句只能在用整型或类似整型的常量标识的case子句中选择，这大大限制了它的使用。在实际编程中，常会用字符串常量或者变量作为case子句的指示符。由于switch语句不能在这些情况下使用，所以只能用级联if语句替代。然而在条件允许的情况下，使用switch语句会使程序更具可读性、更高效。

4.6 while语句

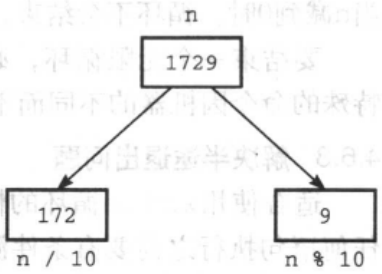
最简单的迭代结构就是while语句，它重复执行一条简单语句或程序块，直到条件表达式变为FALSE为止。while语句的范例如下面的语法框所示。与if语句一样，当循环体只由一条简单语句构成时，C编译器允许去掉加在循环体两边的括号。在本书中，为了增强程序的可读性，在所有while循环体两边都加了括号。 119

语法：while语句

```
while (condition) {
        statements
}
```

其中：*condition*是用来决定循环是否继续开始下一个周期的条件测试。
*statements*是将要重复执行的语句。

这个问题的本质在于如何将表示一个整数的各位数字拆分开来，关键是要想到用/和%这两个算术运算符可以达到目的。不难发现，任何整数的最后一位恰好是它除以10所得到的余数，写成表达式为n%10。而n/10得到的是整数n除了最后一位的其他部分。例如n的值是1729，/和%操作可以将它分成两部分：172和9，如右图所示。



因此为了求这个整数的各位数字的和，需要做的是在循环的每一个周期中将n%10的值加到变量dsum中，再将整数n除以10。下一个周期把原数的十位上的数字加到和中，如此反复直到整数的每一位都处理完毕为止。 121

但是怎样知道何时结束呢？事实上，每个周期中都将n除以10，最终将使n变成0。此时已经处理完该整数的所有数位，也就可以退出循环了。换句话说，只要n的值大于0，就应继续循环。所以，这个程序的while语句应该如下所示：

```
while (n > 0) {
        dsum += n % 10;
        n = n/10;
}
```

完整的digitsum.c程序在图4-5中给出。

```
main()
{
    int n, dsum;

    printf("This program sums the digits in an integer.\n");
    printf("Enter a positive integer: ");
    n = GetInteger();
    dsum = 0;
    while (n > 0) {
        dsum += n % 10;
        n /= 10;
    }
    printf("The sum of the digits is %d\n", dsum);
}
```

图4-5 digitsum.c的主程序

4.6.2 无限循环

当在程序中使用while循环时，一定要确保循环的控制条件最终会变为FALSE，以便能退出循环。如果while的控制行中的条件总是为TRUE，那么计算机将一个周期接着一个周期不停地执行下去，这种情况叫做**无限循环**（infinite loop）。

作为一个例子，假设在程序digitsum.c的while循环中，由于粗心将控制条件中的运算符“>”写成“>=”，如下所示：

```
while (n >= 0) {
    dsum += n % 10;                    (循环将不停地执行下去)
    n /= 10;
}
```

常见错误 仔细斟酌在while循环中使用的条件表达式，确保循环最终会结束。永远不会结束的循环称为**无限循环**。

环颠倒了这两个语句在循环中顺序，使程序较以前难读。其次，范例中要求读入数据的语句出现两次。重复的代码会带来一系列维护上的问题，因为对其中一部分的修改不会改变另外一些副本。很多经验表明，学会使用break语句来解决半途退出问题的学生比那些不会使用break语句的更能正确地编写程序[⊖]。

```
main()
{
    int value, total;

    printf("This program adds a list of numbers.\n");
    printf("Signal end of list with a 0.\n");
    total = 0;
    printf(" ? ");
    value = GetInteger();
    while (value != 0) {
        total += value;
        printf(" ? ");
        value = GetInteger();
    }
    printf("The total is %d\n", total);
}
```

图4-6 修改过后的addlist.c主程序

尽管有上述不利因素，但一些教师还是不喜欢用break来解决半途退出问题。主要是因为这样会导致C语言中break语句的滥用。防止滥用break语句的一个办法就是彻底禁用。有些人认为，这种方式太苛刻了。在本书中，仅在解决while循环的半途退出问题时使用break语句，而在其他容易引起程序结构模糊的情况下不使用break语句。

4.7 for语句

C语言中最重要的一个控制语句就是for语句。它经常在将一个操作重复执行特定次数的情况下使用。下面给出了for语句的一般形式。

语法：for语句

```
for (init; test; step) {
    statements
}
```

其中：*init*是初始化循环时计算的表达式。
*test*是一个条件测试，决定循环是否继续，就如while语句一样。
*step*是为下一个循环周期做准备的表达式。
*statements*是要被重复执行的语句。

for循环的操作是由for控制行中的三个斜体表达式*init*、*test*和*step*决定的。*init*表达式指出for循环应该如何初始化，通常用于设置下标变量的初始值。例如语句

```
for (i=0; . . .
```

⊖ 证实这个发现的最著名的研究是Elliot Soloway、Jeffrey Bonar和Kate Ehrlich合著的*Cognitive Strategies and Looping Constructs: an Empirical Study*（发表于*Communications of the ACM*, November 1983）。

```
10
 9
 8
 7
 6
 5
 4
 3
 2
 1
 0
Liftoff!
```

程序liftoff.c说明任何变量都可以作为下标变量。在这个例子中，下标变量叫做t很可能是因为它表示火箭点火升空前倒数的10秒的变量。下标变量和其他变量一样，在任何情况下都要在程序的开头对它进行声明。

表达式*init*、*test*和*step*中的每一个都是可选的，但它们中间的分号是必需的。如果*init*不出现，则将不执行任何初始化操作。如果*test*不出现，则将假设条件测试为TRUE。如果*step*不出现，则在循环的周期间将不执行任何操作。这样控制行就成了：

```
for ( ; ; )
```

它等效于

```
while (TRUE)
```

4.7.1 嵌套的for循环

当程序变得越来越复杂时，常常需要将一个for循环嵌入到另一个for循环中去。在这种情况下，内层的for循环在外层循环的每一次迭代中都将执行它的所有的周期。每个for循环都要有一个自己的下标变量以避免下标变量间的互相干扰。

图4-8中的程序timestab.c是一个嵌套for循环的一个例子。

```
/*
 * File: timestab.c
 * ----------------
 * Generates a multiplication table where each axis
 * runs from LowerLimit to UpperLimit.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * ---------
 * LowerLimit -- Starting value for the table
 * UpperLimit -- Final value for the table
 */

#define LowerLimit  1
#define UpperLimit 10
```

图4-8 程序timestab.c

可以很清楚地知道循环体中的语句将被执行10次，对应的i值从0～9。等效的while循环是：

```
i=0;
while (i<10) {
    . . . body . . .
    i++;
}
```

其中循环底部的自增操作在循环体很大时很可能会遗漏。

4.7.3 for语句中浮点型数据的使用问题

由于for循环中的*init*、*test*和*step*部分可以是任何表达式，所以没有任何明显的理由要求for循环的下标值必须是整型。在for循环中下标从0～10，每次自增2是完全可以的，代码如下：

```
for (i=0; i<=10; i+=2) . . .
```

这意味着也应该可以将x声明为double型，再将它从1.0起每次自增0.1直到2.0：

```
for (x = 1.0; x <= 2.0; x += 0.1) . . .      (这个测试可能会失败)
```

129

在有些机器上，这条语句可以得到预期结果，而在另一些机器上，可能会得不到最后一个值。例如下面这个for循环：

```
for (x = 1.0; x <= 2.0; x += 0.1) {
    printf("%.1f\n", x);                    (这个循环可能会不显示2.0)
}
```

当它在某些计算机上运行时，会产生以下输出：

```
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
```

注意，从循环控制看来本应显示的值2.0丢失了。

问题出在浮点数并不是百分之百精确的。比如，浮点型值0.1很接近于数学上的分数1/10，但它们并不精确地相等。每当下标变量x加0.1时，误差就被累计，到最后与2.0比较来决定是否结束循环时，x的值可能就成了2.000000001或类似的不是小于或等于2.0的数。这样，for循环的条件不满足，则它在少运行一个周期的情况下就退出了。解决这个问题的最好方法就是限制只能用整型作为for循环的变量。因为整型是精确的，不会造成上述问题。

假如一定要从1.0起累加0.1到2.0，可以采用从10累加1到20，再将它除以10：

```
for (i=10; i<=20; i++) {
     x=i/10.0;
     printf ("%.1f\n", x);
}
```

复习题

1. 下面的结构在C语言中是一个合法语句吗？它有实际用处吗？

```
17;
```

2. 说明下列语句的效果，假设i、j、k声明为整型变量：

```
i = (j = 4) * (k = 16);
```

3. 用哪个简单语句可以将x和y的值设置为1.0（假设它们都被声明为double型）？

4. 结合性是什么意思？与其他运算符相比，赋值号的结合性有什么不一样？

5. 什么是程序块？它的另一个叫法——复合语句说明了程序块的什么重要特征？

6. 控制语句可以分为哪两类？

7. “两个控制语句是嵌套的”这句话是什么意思？

8. bool型数据的两个值是什么？

9. 当一个程序员试图在条件表达式中使用数学中的等号时会发生什么状况？

10. C语言对使用关系运算符来比较的数值的类型有什么限制？

11. 怎样用一个布尔表达式来测试整型变量n的值在0～9之间，并包含0和9？

12. 用自然语言描述下列条件表达式的意思： 132

```
(x != 4) || (x != 17)
```

x取什么值时这个条件为TRUE？

13. 简化求值是什么意思？

14. 假设myFlag声明为布尔型变量，下面的if语句会有什么问题？

```
if (myFlag==TRUE) . . .
```

15. 本书中使用的if语句有哪四种不同的格式？

16. 用自然语言描述switch语句的常见操作。

17. 假设在while语句的循环体中有这样一条语句，当它执行时while循环的条件值就变为FALSE。那么这个循环是将立即终止还是要完成当前周期呢？

18. 在图4-5的程序digitsum.c中，为什么要求整型值为正很重要？

19. 什么是半途退出问题？本书中提到的解决这个问题的两个策略是什么？

20. for语句控制行中出现的三个表达式的作用各是什么？

21. 当遇到下列三个情况时，你将怎样编写for语句的控制行：

a) 从1计数到100。

b) 从0开始，每次计数加7，直到成为三位数为止。

c) 从100开始，反向计数，每次减2，直到为0为止。

22. 为什么在for循环中最好避免使用浮点型变量作为下标变量？

程序设计练习

1. 为了在长途巴士上消磨时间，在美国长大的年轻人常常会反复唱下面这支歌：

99 bottles of beer on the wall.

99 bottles of beer.

You take one down, pass it around.

98 bottles of beer on the wall.

$F_5 = 5$ (2 + 3)

$F_6 = 8$ (3 + 5)

编写一个程序，顺序显示F_0到F_{15}。确保它的输出结果按以下运行示例的方式显示：

```
This program lists the Fibonacci sequence.
 F(0) =    0
 F(1) =    1
 F(2) =    1
 F(3) =    2
 F(4) =    3
 F(5) =    5
 F(6) =    8
 F(7) =   13
 F(8) =   21
 F(9) =   34
F(10) =   55
F(11) =   89
F(12) =  144
F(13) =  233
F(14) =  377
F(15) =  610
```

9. 修改上面的程序，不要指定最后一个数的下标值，而是让程序显示比10 000小的Fibonacci数列。 135

10. 编写程序在屏幕上显示下图所示图案。其中行的数量用#define定义为常量，下面是行数为8的运行示例：

```
*
**
***
****
*****
******
*******
********
```

11. 修改练习10中的程序，让它显示一个不同的三角形。在这个三角形中，每行比上一行多两个星，这个三角形如下所示：

```
       *
      ***
     *****
    *******
   *********
  ***********
 *************
***************
```

136

他要编写一段实现printf函数所需要的程序语句。由于这个工作已经做好了，其他程序员就能使用printf函数，而不需要再编写实现这个功能的语句了，甚至不需要知道实现这个功能的语句是什么。

为了更具体地考虑函数，必须先回顾一下第2章介绍的一些有关函数的基本术语。首先，**函数**（function）是由一组语句组成，并给定一个名字。执行与函数相关的一组语句的行为称为**调用**（calling）这个函数。在C语言中，函数的调用方法是写一个函数名，后面跟一组括在 138
一个圆括号中的表达式。这些表达式称为**实际参数**（argument），是调用程序传给函数的信息。在hello.c中，由于主程序将所需的数据作为函数调用的一部分，因此printf函数知道要显示什么内容。如果函数不需要从它的调用程序那里获取信息，就不需要实际参数，但是函数调用中必须要有一对空的圆括号。

一旦被调用，函数就从实际参数中获取数据，完成相应的工作，然后返回调用它的程序点。注意，调用程序所做的工作以及如何精确地返回到调用点是函数调用机制所定义的特性之一。返回到调用程序的操作称为从函数**返回**（returning）。作为返回操作的一部分，函数也能将结果返回给调用程序，如在语句

```
n1 = GetInteger();
```

中的函数GetInterger所示。在GetInteger函数完成了从用户处读取一个整数的任务后，它把这个整数作为GetInteger()调用的值返回给调用程序。这个操作被称为**返回一个值**（returning a value）。

从某种意义上讲，参数提供了函数的输入，返回值是它的输出，输出返回给调用程序。尽管在概念上是非常类似的，但严格区分输入操作（如GetInteger）和在函数作用域中使用实际参数还是很重要的。函数（如GetInteger）提供了从用户获取输入的机制。当GetInteger需要一个输入值时，终端的用户必须在键盘上输入这些值。而函数的实际参数则提供了一种使函数从它的调用程序那里获取输入值的方法，它是从程序的另一部分而不是从用户那里获取。以实际参数形式传递的数据必须由用户在程序前面的某一地方输入，并可能作为程序的某一部分计算过。也必须仔细区分输出操作（如printf）的使用和返回一个结果的技术。当使用printf时，输出显示在终端屏幕上。当函数返回一个结果时，该信息回到了调用程序，调用程序可以以任何方式使用该结果。当需要用实际参数和返回值的时候，某些新程序员会倾向于在函数内使用输入/输出操作。

常见问题 要仔细区分程序中的输入输出和函数中的实际参数和返回值的概念。输入输出允许**程序**和它的**用户**通信。参数和返回值允许**函数**和它的**调用程序**之间通信。

为了理解如何在C语言中使用函数，必须知道函数调用是一个简单的表达式，它可以出现在任何表达式可以出现的地方。而且，函数的实际参数也是一个表达式，它本身也可包含函数调用或任何合法表达式中的操作。

为了说明函数和它的实际参数是表达式，我们介绍一些math系统库中的一些标准函数。这个库包含了许多代数和三角中的标准数学函数。例如，math库包括取它实际参数的平方根的函数sqrt，以及正弦和余弦函数sin和cos。这些函数都有double类型的实际参数和返 139
回值。可以把这些函数用在简单的语句中，如

```
root3 = sqrt(3.0);
```

根。然而，C编译器不需要这个信息。它必须知道的只是sqrt取一个double类型的值，并返回一个double类型的值。函数的确切的作用是通过函数名和相关的文档告诉程序员的。

为程序员提供有用信息的另一个途径是为每一个参数提供一个描述性的名字，该名字可标识特定参数的作用（还可以包含相应的参数规格说明）。虽然参数的名字为使用该函数的程序员提供了重要的信息，但对程序没有任何实质性的影响。例如，在math.h中sin被声明为

```
double sin(double);
```

它仅指出了参数的类型。需要用这个函数的程序员可能更希望看到这个原型被写为

```
double sin(double angleInRadians);
```

以这种形式编写的原型提供了一些有用的新信息：sin函数有一个double类型的参数，该参数是以弧度表示的一个角度。在你自己的函数中，你应该为参数指定名字，并在相关的介绍函数操作的注释中使用这些名字。

如果函数没有参数，C用特殊的关键词void作为参数规格说明。例如，simpio库中的函数GetInteger没有从它的调用程序那里获取参数，返回的是一个int类型的值。这个函数的原型可以写为

```
int GetInteger(void);
```

5.3 自己编写函数

到目前为止，本章所介绍的函数都是某些程序库的一部分。库函数只对它们的正确性感兴趣，不会说明函数的全部内容。作为程序员，通常只要使用库函数，而不必知道任何内部细节。但你也有可能会使用不属于任何库的函数。在这些情况下，除了定义你自己的函数之外别无选择。

例如，你接受了一个任务，写一个程序将大多数国家使用的摄氏温度转换为美国使用的华氏温度。你可以定义一个简单的转换函数，它可以用在程序的其他部分。这个计算是相当简单的，将温度从一种计量标准转换为另一种计量标准只要使用下列公式即可。 141

$$F=\frac{9}{5}C+32$$

在C程序中增加一个新的函数由两个步骤组成：

1) 需要指定这个函数原型，它通常位于整个程序的头部，在#include行之后。

2) 在程序的稍后部分，需要提供该函数的实现，即指定一些实际的程序步骤。

原型是很短的，仅指出了函数的实际参数和返回类型。实现则较长，它要提供实现的细节。

在编写一个函数时，通常最好从编写它的原型开始。在这个例子中，你要写一个将摄氏温度转为华氏温度的函数。你可以用第2章中命名变量的一些规则，为函数选名字。这个名字要使得读程序的人容易确定函数要做什么。例如，下面的名字就不容易造成误解：

```
double CelsiusToFahrenheit(double c);
```

函数的实现是由它的原型开始，去掉结尾的分号，然后加入函数体。**函数体**（function body）是一个程序块，由一组括在一对花括号中的语句组成。程序块中的语句前面可以有变量声明，就如前面所见的那些main函数中一样。

5.3.1 return语句

如果函数返回一个结果，则函数体中至少包含一个**return语句**，它指出了要返回的值。

```
 * UpperLimit -- Final value for temperature table
 * StepSize   -- Step size between table entries
 */

#define LowerLimit   0
#define UpperLimit 100
#define StepSize     5

/* Function prototypes */

double CelsiusToFahrenheit(double c);

/* Main program */

main()
{
    int c;

    printf("Celsius to Fahrenheit table.\n");
    printf("  C    F\n");
    for (c = LowerLimit; c <= UpperLimit; c += StepSize) {
        printf("%3d  %3g\n", c, CelsiusToFahrenheit(c));
    }
}

/*
 * Function: CelsiusToFahrenheit
 * Usage: f = CelsiusToFahrenheit(c);
 * ----------------------------------
 * This function returns the Fahrenheit equivalent of the Celsius
 * temperature c.
 */
double CelsiusToFahrenheit(double c)
{
    return (9.0 / 5.0 * c + 32);
}
```

143
~
144

图 5-1 (续)

主程序由一个第4章中用来生成表的for循环组成。温度转换表中的每一行都是由语句

```
printf("%3d %3g\n", c, CelsiusToFahrenheit(c));
```

生成。这个语句调用CelsiusToFahrenheit函数计算对应于摄氏温度c的华氏温度[⊖]。结果值作为实际参数传递给printf，printf继续显示该值。

注意，图5-1的程序中有许多注释，比以前出现过的程序都多。每个函数也应该有它自己的描述性的注释，这样，程序的读者可以将每个函数作为一个单元来理解。以我的经验，为函数写的最有用的注释之一是给出一个函数是如何被使用的实例。在本书中，函数的注释都包含了一个提供这种实例的“Usage”行。

5.3.3 包含内部控制结构的函数

函数并不是都像CelsiusToFahrenheit那么简单。在许多情况下，计算一个函数需要

⊖ 在for循环中，下标变量c声明为int型，尽管CelsiusToFahrenheit被定义为取一个double的参数。正如4.7.3节指出，整型数是精确的，因此可保证程序运行正确的次数。当用参数c调用CelsiusToFahrenheit时，通过自动类型转换将参数值转换为double类型。

1) 定义一个变量product。

2) 把它初始化为1。

3) 将1～n之间的整数逐个与它相乘。

4) 将product的最终值作为函数的结果返回。

为了得到第三步中所需的每一个整数，需要一个for循环，它从1开始一直变化直到n为止。for循环需要一个下标变量，通常用i表示。因此Factorial需要声明两个变量：

```
int product, i;
```

变量product保存运行过程中的乘积值，i是下标变量。

如图5-2所示，Factorial的实现非常简单，不需要详细地解释细节问题了。

```
int Factorial(int n)
{
    int product, i;

    product = 1;
    for (i = 1; i <= n; i++) {
        product *= i;
    }
    return (product);
}
```

图5-2 Factorial函数

5.3.4 返回非数字值的函数

至今为止所介绍的函数实例返回的都是数字值，函数这个词与数学上的历史渊源使大家很容易想到创建数值函数。但在C语言中，函数可以返回任意数据类型的值。例如，要编写一个对日期进行操作的程序，那么有一个将用数值表示的月份（1～12）转换为相应的用文字表示月份名（从January到December）的string类型的函数是非常有用的。数值型的值内部工作较容易（例如，比较两个月份，看哪一个较早一些），但用文字输出月份名会使可读性更强一些。为了解决这个问题，可以定义一个函数MonthName，如图5-3所示。

```
string MonthName(int month)
{
    switch (month) {
      case  1: return ("January");
      case  2: return ("February");
      case  3: return ("March");
      case  4: return ("April");
      case  5: return ("May");
      case  6: return ("June");
      case  7: return ("July");
      case  8: return ("August");
      case  9: return ("September");
      case 10: return ("October");
      case 11: return ("November");
      case 12: return ("December");
      default: return ("Illegal month");
    }
}
```

图5-3 MonthName函数

为了使用这个函数，在程序的其他部分应该调用MonthName，然后用printf显示这个结果。例如，整数变量month、day和year的值分别为7、20、1969（阿波罗11登上月球的日子），语句

```
printf("%s %d, %d\n", MonthName(month), day, year);
```

会产生一个输出

146
～
147

如下：

```
bool IsLeapYear(int year)
{
    return (((year % 4 == 0) && (year % 100 != 0))||(year % 400 ==0));
}
```

在第4章中，已经介绍过一个确定year是否为闰年的布尔表达式。把这个表达式放入一个函数，就不需要在测试时明确地包含这个完整的计算。一旦这个函数被定义，在程序的其他地方可以用如下语句调用函数：

```
if (IsLeapYear(year)) . . .
```

5.3.6 测试字符串是否相等的谓词函数

到目前为止，你可以对字符串数据做一些简单的操作，例如用GetLine读入字符串，用printf将字符串显示在屏幕上。在第9章中，你将学到完整的字符串函数库，有了这个函数库，可以对字符串数据进行各种操作。这里将介绍这些函数中的一个。这个函数就是StringEqual，你可以用这个函数区分两个字符串是否包含完全相同的字符。

在这里介绍StringEqual函数有两个主要原因。首先，在程序设计习语中使用StringEqual使你能以更有创造性的方法使用字符串，从而写出一些更有趣的程序。其次，StringEqual是一个谓词函数，这样有助于说明谓词函数在程序设计应用中的重要性。 149

StringEqual函数的原型为：

```
bool StringEqual (string s1, string s2);
```

它指出StringEqual函数以两个字符串作为实际参数，并返回一个布尔值。如果两个字符串s1和s2中的字符完全相同，这个值为TRUE。如果两个字符串中的字符有一点点不同，StringEqual返回FALSE。下面的例子说明了这种行为：

```
StringEqual ("abc", "abc")      返回TRUE
StringEqual ("abc", "def")      返回FALSE
StringEqual ("abc", "abcd")     返回FALSE
StringEqual ("abc", "ABC")      返回FALSE
```

注意，字符串中的字符必须完全相同。如果一个字符串中多了一个字符或两个字符串中的字符稍有差别，StringEqual返回FALSE。

在需要问用户一个问题并根据回答采取某些行动时，可以使用StringEqual函数。例如，假设编写了一个与用户做游戏的程序，你要给用户提供一个重玩的机会。你可以在屏幕上提供如下适当的提示，向用户提出问题，然后调用GetLine获取响应：

```
printf("Would you like to play again?");
answer=GetLine();
```

StringEqual函数使得程序可以针对响应完成某些操作。例如，你可以写一个如下所示的主程序：

```
main()
{
    string answer;

    while (TRUE) {
        PlayOneGame();
```

```
/*
 * Function: Factorial
 * Usage: f = Factorial(n);
 * ------------------------
 * Returns the factorial of the argument n, where factorial
 * is defined as the product of all integers from 1 up to n.
 */
int Factorial(int n)
{
    int product, i;

    product = 1;
    for (i = 1; i <= n; i++) {
        product *= i;
    }
    return (product);
}
```

图 5-4（续）

如果把fact.c程序看成两个部分，该程序很有意义。主程序只是从LowerLimit到UpperLimit计数。在每一次循环中，对下标变量i的值调用Factorial函数，并显示函数结果。现在，我们通常用名字i表示用于计算for循环的循环次数的下标变量。不要感到惊奇。Factorial函数同样很直观。它比主程序复杂一点。你很容易理解函数要做什么。特别是，你可以看出n是我们要计算阶乘的数，product保存每次循环的乘积值，i是一个没有特别意义的下标变量，用来跟踪for循环的进展。这样，我们清楚了程序的每个片断的意义。

对一些新手来说，只有当把程序看成一个整体时才会糊涂。如果你读一个程序但没有理解如何看待函数，可能会出现一些有趣的问题。首先，有两个名为i的变量，一个在主程序里，一个在Factorial函数中。每个变量都作为下标变量，但这两个变量有不同的值。第二个也同样会混淆的问题是在程序的其他部分，其中两个不同的名字用来指向同一个值。在主程序中，我们要计算阶乘的数被存放在下标变量i中。在Factorial函数中，同一个值被称为n。如何搞清楚这些问题？

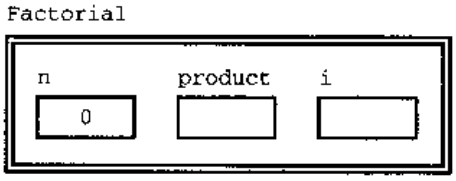
虽然把程序作为整体来看会产生混淆，但要记住当每次只看一个函数时，该函数是有意义的。如果只看main或Factorial，这些令人混淆的问题都不存在。每个函数定义只使它自己有意义。事实上，越是常见的越重要。随着程序规模的增长，没有办法将它们作为一个整体来理解。要理解大的程序，只有把它分块，每一分块足以使得它自己有意义。在fact.c程序中，求一个阶乘表的问题被分解为两个容易理解的分块。

5.4.1 参数传递

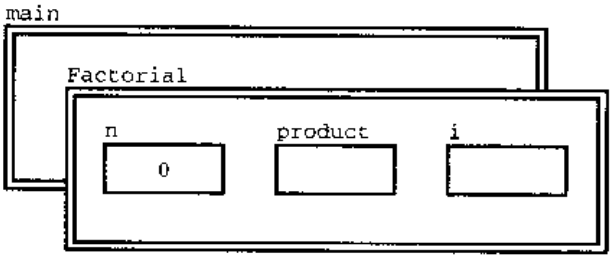
为了理解程序的两个部分是如何协同工作的，先要理解C语言本身如何处理这些混乱问题。对程序员来说，了解这个处理方法是很重要的，这使得我们可以单独考虑每个函数。C语言是如何使同一个名字有不同的值？又如何使得同一个概念值用不同的名字表示？为了理解这个问题的答案，有必要介绍一下函数调用中的实际参数值和函数中用来保存这个值的变量在语义上的区别。

当主程序调用Factorial（它本身是作为函数printf的实际参数）时，实际参数是表达式i。当这个语句被执行时，其效果相当于检查i的当前值，并把它传给函数Factorial，

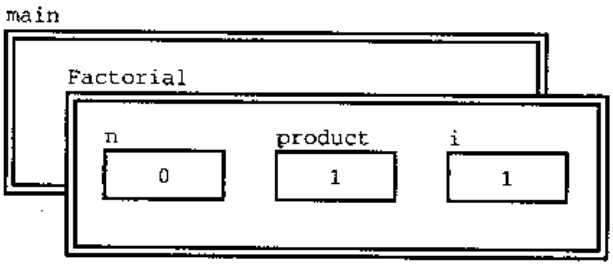
0）进行初始化。则帧的内容如右图所示。当为Factorial创建一个新帧时，main的帧没有完全去掉。该帧暂时被放在一边，直到Factorial的操作结束为止。为了在用方框图表示的概念模型上表示这种情况，最好将每个新帧图画在一张索引卡上，然后将新帧放在表示原来的帧的卡上，即覆盖在它上面。例如，当调用函数Factorial时，表示Factorial帧的索引卡放在了main帧上面。



如下图所示，整个帧集合形成了一个栈，最近的帧放在最上面，这就是栈帧这个术语的来源。一旦Factorial被激活，main的帧仍然存在，只是你看不到它的任何内容。特别要注意，名字i不再引用main函数中声明的变量，而只引用Factorial中的变量名i。

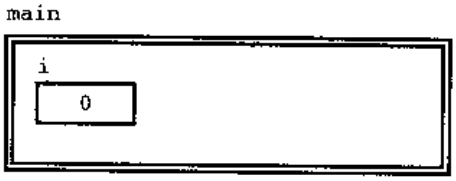


下一步是执行Factorial的函数体，方法是在当前的帧上运行函数的每一步。变量product被初始化为1，程序到达for循环。在for循环中，变量i被初始化为1，但由于该变量值已经大于n，所以for循环体没有被执行。因此，当程序到达return语句时，帧的内容变为：



155

当Factorial返回时，product的值作为函数的结果值传回调用程序。从函数返回也意味着扔掉它的帧，使main中的变量重新显现出来，如右图所示。然后，结果值1被传给printf函数。printf函数经历同样的过程。然而，操作的细节用户是不可见的，因为用户不知道printf在内部是如何工作的。最终，printf将结果值1显示在屏幕上，然后返回，之后main函数继续执行for循环的下一周期。



5.4.2 在其他函数中调用函数

函数的主要作用在于一旦一个函数被定义，不仅主程序可以使用它，而且可以作为实现其他函数的工具，使得这些函数可以用作为更复杂的工具。这些函数再被其他函数调用，以此类推，从而创建出具有任意复杂度的层次结构。

假设在你前面的桌上有n个不同物体组成的一个组。从这组物体中选出k个物体。要解决的问题是：从n个不同物体的集合中选出k个物体有多少种不同的方法？

为了使问题更加具体，假设桌上的物体是五个美元硬币：1美分、5美分、1角、25美分和

```
    printf ("Enter number of objects in the set (n)?");
    n = GetInteger ();
    printf ("Enter number to be chosen (k)?");
    k = GetInteger();
    printf ("C(%d, %d) = %d\n", n, k, Combinations (n,k));
}
```

完整的程序combine.c如图5-5所示。

```
/*
 * File: combine.c
 * ---------------
 * This program tests a function to compute the mathematical
 * combination function Combinations(n, k), which gives the
 * number of ways to choose a subset of k objects from a set
 * of n distinct objects.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Function prototypes */

int Combinations(int n, int k);
int Factorial(int n);

/* Main program */

main()
{
    int n, k;

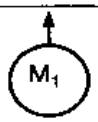
    printf("Enter number of objects in the set (n)? ");
    n = GetInteger();
    printf("Enter number to be chosen (k)? ");
    k = GetInteger();
    printf("C(%d, %d) = %d\n", n, k, Combinations(n, k));
}
/*
 * Function: Combinations
 * Usage: ways = Combinations(n, k);
 * ---------------------------------
 * Implements the Combinations function, which returns the number
 * of distinct ways of choosing k objects from a set of n objects.
 * In mathematics, this function is often written as C(n,k), but a
 * function called C is not very self-descriptive, particularly in
 * a language which has precisely the same name.
 */

int Combinations(int n, int k)
{
    return (Factorial(n) / (Factorial(k) * Factorial(n - k)));
}
```

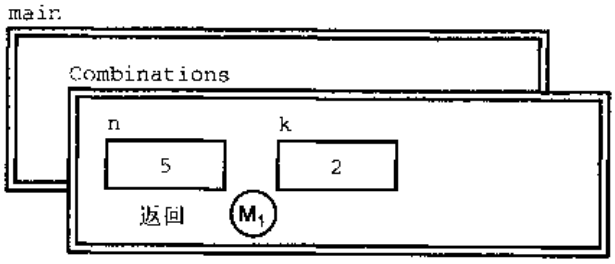
158

图5-5 combine.c

```
    printf("Enter number to be chosen (k)? ");
    k = GetInteger();
    printf("C(%d, %d) = %d\n", n, k, Combinations(n, k));
}
```

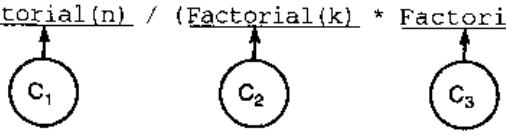


当计算机执行一个新的函数调用时，它记录了一旦调用完成后调用程序应该从哪里继续执行。执行的继续点被称为**返回地址**（return address），在这些图中用一个圆形标记指出。为了记住程序执行的位置，应该在帧图上记录调用点，如： 160

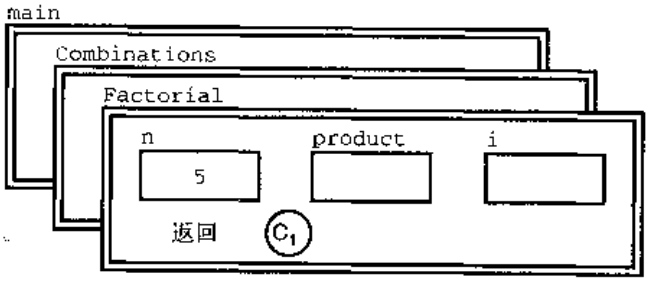


一旦创建新的帧，程序开始执行Combinations函数的函数体，它将用标记记录每个Factorial函数的调用：

```
int Combinations(int n, int k)
{
    return (Factorial(n) / (Factorial(k) * Factorial(n - k)));
}
```



为了执行这个语句，计算机必须做出三个调用函数Factorial的标记。按照ANSAI C制定的标准，编译器可以以任何次序做出这些调用。在这种情况下，假设选择按从左到右的顺序计算。第一个调用请求计算机计算Factorial(n)，这将创建如下的新帧：



Factorial函数按前面已介绍过的过程运行，并返回值120到标记C_1指出的调用点。Factorial函数的帧就消失了，回到Combinations的帧中，继续下一步计算。可以用返回并将调用结果填在调用的位置的方法来说明当前的状态，如下所示： 161

```
return (120 / (Factorial(k) * Factorial(n - k)))
```



图中120外面的方框指出这个值不是程序的一个部分，而是一些以前计算的结果。

```
C(5, 2) = 10
```

这是该程序的最后一个操作。

这个深入剖析内部细节的练习是为了帮助你理解C语言中的函数调用机制。你将发现利用这种方法将自己的程序检查一遍或两遍是很有帮助的，但不要把它当成一种习惯。相反，你应该学会用非形式化的方法考虑函数，尝试从直观上理解函数是如何工作的。当程序调用一个函数时，函数执行它自己的操作，然后程序从调用点继续执行。如果函数返回一个结果，调用程序在以后的计算中可以自由应用此结果。作为一个程序员，你应该找到一个点，在此你可以舒服地考虑这个过程，而不需要为细节问题而烦恼。毕竟，计算机会为你考虑这些问题。

5.5 过程

到目前为止，本章中的函数都为它们的调用程序返回某种类型的值。在程序设计中，有时调用一个函数不是为了得到它的返回值，而是要它产生某些作用，这种用法是相当普遍的。 163
例如，当你在一个语句中用以下方法调用printf函数时，

```
printf("Hello world.\n");
```

你并不希望让printf返回某些值。你所关心的是在屏幕上显示某些信息。

不返回值的而是起到某些作用的函数被称为过程（procedure）。许多程序设计语言，包括Pascal和Fortran，提供了完全不同的机制定义函数和过程。在这些语言中，函数和过程是两个不同的概念实体。在C语言中，这两个概念是合在一起的，大多数有关C语言的书都用术语函数表示函数和过程。在本书中，我们基本遵循这个习惯。但认识到某些函数是不返回结果的，这很重要。因此，当强调某个函数没有返回值时，本书会使用过程这个词。

在C语言中，通过在函数原型中返回类型的地方用关键词void来表示过程。例如，原型

```
void GiveInstructions (void);
```

声明一个没有参数和返回值的函数。对任何函数来讲，原型出现在程序的开始部分；而GiveInstructions的实现则出现在程序后面的部分中。这个过程的一般的实现如下所示：

```
void GiveInstructions(void)
{
    printf ("This program performs some important calculation\n");
    printf ("for the user. This function, or one very much\n");
    printf ("like it, can be used to give the user whatever\n");
    printf ("instructions are required to use the program,\n");
    printf ("such as the format for input data and the like.\n");
}
```

与大多数过程一样，GiveInstructions没有包括return语句，而是直接“跌到了终点”，C语言解释这种情况为过程的执行完成了。然而，过程也可以用return语句强制立即从过程返回。当return语句用在过程中时，单词return后面没有表达式。

一旦定义GiveInstructions过程，主程序为了在屏幕上依次显示这些指令时，就要调用这个过程。因此可假定主程序从下列语句开始：

解的特殊情况或限制下结束时更是如此。逐步精化的原则指出：一旦你有了某个程序的概要描述，你就应该在此结束，并把它写下来，调用过程或函数表示那些还要继续细化的程序。如果采用这种方法，主程序可以写成：

```
main()
{
    int year;

    GiveInstructions();
    Year =GetYearFromUser();
    PrintCalendar(year);
}
```

第一个语句调用GiveInstructions过程，该过程的实现与本章前面的5.5节中出现的GiveInstructions过程非常类似。第二个语句调用函数GetYearFromUser，该函数可用来处理读入年份的过程。GetYearFromUser的实现可以简单地表示为显示一个提示符，并调用GetInteger，但也可以包括一些其他的操作，如检查用户输入的年份是否合法。主程序的最后一行调用函数PrintCalendar，给它传递所需的年份作为参数。最后一个函数的实现将涉及解决整个问题的绝大部分工作，但这个问题现在已经变得稍微简单一些了，因为主程序已经考虑了所有与用户的交互。

在main实现所示的详细级别上，程序已经具有完整的意义。只要GiveInstructions、GetYearFromUser和PrintCalendar完成它们的工作，程序就应该正常工作。当然，你还没有编写这些函数的实现，因此现在就依赖于他们似乎太早。然而，上述工作是成功分解的关键。只要沿着这条路进行下去，你会发明新的过程和函数来解决这些任务中的新的有用的片段，然后，根据这些片段实现解决方案层次结构中的每一层。 166

5.6.2 实现PrintCalendar

写完了主程序之后，还要编写三个函数。虽然你可以从编写任何一个函数开始，但我们直接进入问题的中心，实现PrintCalendar。从main程序的调用判断，PrintCalendar的原型是

```
void PrintCalendar (int year);
```

它的实现怎么写？

在抽象级别考虑PrintCalendar函数，考虑它要做些什么。它通过打印12个单独的月历完成日历的显示。利用这些想法足以编写一个适合这个分解层次的该函数的实现。PrintCalendar的函数体是一个简单的循环，它调用另外一个函数显示某一个月的月历，然后打印一个换行符使得该月与下一个月之间有一个空行：

```
void  PrintCalendar(int year)
{
    int month;

    for (month = 1; month <= 12; month++) {
        PrintCalendarMonth(month, year);
        Printf ("\n");
    }
}
```

PrintCalendarMonth函数有两个参数，月份和年份，因为这两个信息它都需要。参数month

用在这里就不太合适，weekday最终会成为7、8、9等值，这些值没有对应的星期几。通过除7取余可以保证结果值总是在0～6之间。当采用取余数操作将计算的结果限制在一个较小的周期性的范围内时，使用的是数学上称为**取模运算**（modular arithmetic）的过程。取模运算在程序设计中非常有用，在本书中还可以看到一些其他实例。

如果你保存了星期几的记录，那么在PrintCalendarMonth函数中写一个主循环并不困难。下面的代码就能完成这些工作：

```
for (day = 1; day <= nDays; day++) {
    printf("%2d", day);
    if (weekday == Saturday) printf("\n");
    weekday = (weekday + 1) % 7;
}
```

该循环显示了每一个数，记录了它是星期几，在每个星期六之后换行。日历的最后一行必须以换行结束，因此循环后必须紧跟以下的语句：

```
if (weekday != Sunday) printf ("\n");
```

这样就保证即使这个星期不是在星期日结束，最后一行后面也有换行符。

至此，还剩下三个任务：

1) 计算出该月的天数。

2) 确定该月的第一天是星期几。

3) 缩排月历的第一行，使得第一天出现在正确的位置。

逐步精化的策略建议不要在分解的这个层次上解决这三个问题。事实上，可以将这三个操作转化为函数调用，这些函数可以以后再实现。应用这个策略，可以写一个完整的PrintCalendarMonth的实现：

```
void PrintCalendarMonth(int month, int year)
{
    int weekday, nDays, day;

    printf("    %s %d\n", MonthName(month), year);
    printf(" Su Mo Tu We Th Fr Sa\n");
    nDays = MonthDays(month, year);
    weekday = FirstDayOfMonth(month, year);
    IndentFirstLine(weekday);
    for (day = 1; day <= nDays; day++) {
        printf(" %2d", day);
        if (weekday == Saturday) printf("\n");
        weekday = (weekday + 1) % 7;
    }
    if (weekday != Sunday) printf("\n");
}
```

169

用逐步精化的方法实现PrintCalendarMonth意味着将其中用到的三个函数（MonthDays、FirstDayOfMonth和IndentFirstLine）的实现将放到以后再写。当你回过头来填补这些缺少的部分时，你可以按任意次序来实现这些函数，但在你执行程序前必须编写完成所有的函数。

最容易实现的函数是这组函数中的最后一个：IndentFirstLine。这个函数从FirstDayOfMonth中获取一个星期中的第几天，保证月历的第一行前有足够的空格，使第一天出现在正确的位置。如果这个月从星期天开始，月历将在第一行的起始位置开始。如果从星期一开始，程序将打印出表示一天所需的空格来表示缺少星期天。因为每个月历上的项目占用3

```
        weekday = (weekday + MonthDays(i, year)) % 7;
    }
    return (weekday);
}
```

5.6.4 完成最后的片段

完成这个程序只需要填充一些细节。例如，需要实现函数GiveInstructions和GetYearFromUser。FirstDayOfMonth的定义要求处理的年份不能早于1900年，因此最好能在这些函数中做一些限制，强迫满足这个要求。例如，下面的GetYearFromUser的实现检查输入的年份是否满足这个条件，如果不满足，则给用户一个重新输入的机会。

```
int GetYearFromUser(void)
{
    int year;

    while (TRUE) {
        printf("Which year? ");
        year = GetInteger();
        if (year >= 1900) return (year);
        printf("The year must be at least 1900.\n");
    }
}
```

171

剩下的工作还有需要从本章前面拷贝IsLeapYear和MonthName函数的实现，确保包括所有的函数原型、写注释、最终编译和测试程序。完整的实现如图5-6所示。

```
/*
 * File: calendar.c
 * ----------------
 * This program is used to generate a calendar for a year
 * entered by the user.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants:
 * ----------
 * Days of the week are represented by the integers 0-6.
 * Months of the year are identified by the integers 1-12;
 * because this numeric representation for months is in
 * common use, no special constants are defined.
 */

#define Sunday     0
#define Monday     1
#define Tuesday    2
#define Wednesday  3
#define Thursday   4
#define Friday     5
#define Saturday   6
```

图5-6 calendar.c

```
/*
 * Function: PrintCalendar
 * Usage: PrintCalendar(year);
 * ---------------------------
 * This procedure prints a calendar for an entire year.
 */

void PrintCalendar(int year)
{
    int month;

    for (month = 1; month <= 12; month++) {
        PrintCalendarMonth(month, year);
        printf("\n");
    }
}

/*
 * Function: PrintCalendarMonth
 * Usage: PrintCalendarMonth(month, year);
 * ---------------------------------------
 * This procedure prints a calendar for the given month
 * and year.
 */

void PrintCalendarMonth(int month, int year)
{
    int weekday, nDays, day;

    printf("    %s %d\n", MonthName(month), year);
    printf(" Su Mo Tu We Th Fr Sa\n");
    nDays = MonthDays(month, year);
    weekday = FirstDayOfMonth(month, year);
    IndentFirstLine(weekday);
    for (day = 1; day <= nDays; day++) {
        printf(" %2d", day);
        if (weekday == Saturday) printf("\n");
        weekday = (weekday + 1) % 7;
    }
    if (weekday != Sunday) printf("\n");
}

/*
 * Function: IndentFirstLine
 * Usage: IndentFirstLine(weekday);
 * --------------------------------
 * This procedure indents the first line of the calendar
 * by printing enough blank spaces to get to the position
 * on the line corresponding to weekday.
 */

void IndentFirstLine(int weekday)
{
    int i;
```

174

图 5-6（续）

```
string MonthName(int month)
{
    switch (month) {
      case  1: return ("January");
      case  2: return ("February");
      case  3: return ("March");
      case  4: return ("April");
      case  5: return ("May");
      case  6: return ("June");
      case  7: return ("July");
      case  8: return ("August");
      case  9: return ("September");
      case 10: return ("October");
      case 11: return ("November");
      case 12: return ("December");
      default: return ("Illegal month");
    }
}

/*
 * Function: IsLeapYear
 * Usage: if (IsLeapYear(year)) . . .
 * ----------------------------------
 * This function returns TRUE if year is a leap year.
 */

bool IsLeapYear(int year)
{
    return ( ((year % 4 == 0) && (year % 100 != 0))
             || (year % 400 == 0) );
}
```

图 5-6 (续)

小结

在本章中，我们介绍了函数，它使你能用一个简单的名字表示一组完整的操作。利用函数时，程序员可以忽略内部细节，而将注意力集中于函数的整体作用，所以函数提供了减小程序复杂性的关键的工具。

176

本章介绍了最基本的程序设计技术——*逐步精化*的方法。从程序的概要描述开始，通过把概要中的每一片段再划分成更简单的步骤来不断精化每一片段，这样可以通过编写一组函数（其中每一个都相当简单）来解决一个复杂问题。

本章介绍的重点内容包括：

- 函数是由一组程序语句组成，这些语句被放在一起，并被赋予一个名字。程序的其他部分可以调用这些函数，以实际参数的形式传给它一些信息，得到一个由函数返回的结果。
- 每个函数在使用前必须被声明。函数的声明被称为*原型*。
- 除了原型之外，函数还有一个实现，它指出了函数包含的每一个步骤。
- 返回一个值的函数必须有一个`return`语句，它指出了函数的结果。函数可以返回任何类型的值。

的值。常量ϕ被称为黄金比例。古代数学家认为这个比例是从美学上看矩形两条边的最佳比例，但它也经常出现在计算数学中。 178

2. 在中学代数中，我们已经学过标准的二次方程式

$$ax^2+bx+c=0$$

可以用公式

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

求得两个根。第一个根是用“+”替换公式中的“±”，第二个根是用“-”替换公式中的“±”。

写一个C语言程序输入a、b和c三个值，然后计算两个根。如果平方根里面的值是负数，则该方程没有实数解，程序要显示这个信息。可以假设a的值为非0。程序的运行示例如下：

```
Enter coefficients for the quadratic equation:
a: 1↵
b: -5↵
c: 6↵
The first solution is 3
The second solution is 2
```

3. 修改图5-1所示的程序c2ftable.c，使它能将华氏温度转换为摄氏温度。你的程序应该被分为主程序和转换函数，就如c2ftable.c程序一样。在主程序中，华氏温度的范围为32～100，步长为2度，以此产生一个等价的摄氏温度表，小数点后保留一位。输出表的基本格式如下所示：

```
Fahrenheit to Celsius table.
  F     C
 32    0.0
 34    1.1
    ...
 98   36.7
100   37.8
```

4. 第4章的练习8中引入了Fibonacci数列，其中每一项是前两项的和。按该练习的条件重写程序，但要改变实现方式，使主程序调用一个函数Fib(n)计算Fibonacci数列的第n项。若所需计算的次数相同，比较新的实现方法和第4章所用方法哪个效率更高？ 179

5. 写一个函数RaiseInToPower，它取两个整数，n和k，返回n^k。用你设计的函数显示一张2^k的表，k从0～10。

6. 写一个函数RaiseRealToPower，它取一个浮点数x和一个整数k，返回x^k。实现你的函数使它在k为负数时也能正确计算结果，用关系式

$$x^{-k}=\frac{1}{x^k}$$

用你的函数显示所有k为－4～4之间的10^k的表，如下表所示：

使得循环正确地显示值1.0、1.1、1.2等，包括2.0。如4.7.3节指出，由于浮点数精度的限制，在某些机器上这个循环将不会包括值2.0。

11. 在5.3.6节中，下面的循环用来给用户再玩一次游戏的机会（游戏本身没有指定）：

```
while (TRUE) {
    PlayOneGame();
    printf("Would you like to play again? ");
    answer = GetLine();
    if (StringEqual(answer, "no")) break;
}
```

181

该方法的主要问题是仅当用户输入的是no时，循环才会退出。程序将其他的响应（如NO或Absolutely not）解释为TRUE，并重新开始游戏。从程序设计的观点来讲，将用户的响应与yes和no比较会更好一些。如果用户给出了两个答案中的一个，程序按答案执行。如果用户给出另外的响应，程序会重新问这个问题。

写一个谓词函数GetYesOrNo，它接收一个提示符串作为参数，将这个提示符显示给用户作为问题。如果用户用单词yes响应，GetYesOrNo将返回TRUE。如果用户用no响应，GetYesOrNo将返回FALSE。如果用户给出任何其他响应，函数会显示一个消息，告诉用户合法的响应值，然后重新提该问题。

给出GetYesOrNo函数，可以将玩游戏的循环改变为：

```
while (TRUE) {
    PlayOneGame();
    if (!GetYesOrNo("Would you like to play again? ")) break;
}
```

它会产生下面的运行示例：

```
. . . play the game . . .
Would you like to play again? yes↵
. . . play the game . . .
Would you like to play again? maybe↵
Please answer yes or no.
Would you like to play again? yes↵
. . . play the game . . .
Would you like to play again? no↵
```

12. 本书中Combinations函数的值通常以下图所示的三角形形式输出

C(0,0)

C(1,0) C(1,1)

C(2,0) C(2,1) C(2,2)

C(3,0) C(3,1) C(3,2) C(3,3)

C(4,0) C(4,1) C(4,2) C(4,3) C(4,4)

这个图称为Pascal三角形，以发明它的17世纪法国数学家Blaise Pascal命名。Pascal三角形有一个有趣的特性，它的每一个内部项是它上面两个项之和。

182

写一个C语言的程序，显示如下所示的Pascal三角形的前8行：

第6章 算 法

People like definite decisions,
Tidy answers, all the little ravellings
Snipped off, the lint removed. . . .

— Gwendolyn Brooks, *Annie Allen*, 1949

学习目标

- 通过研究经典数学中的一些算法，深入理解算法形成过程。
- 了解大多数问题都是可以使用不同的算法来解决。
- 掌握在不同算法中选取某个算法时需要考虑的问题。
- 掌握如何确定算法是正确的。
- 理解并能应用brute-force算法。
- 能够用连续逼近的技术编写程序。
- 学会用`Error`函数向用户报告错误情况。
- 能够编写一个完成级数展开的程序。

184
~
185

函数对于程序设计来说是很重要的，其中的一个原因是函数提供了算法实现的基础。算法本身是抽象的策略，通常用自然语言表达。函数是以某种程序设计语言表示的算法的具体实现。当你要将算法作为程序的一部分实现时，通常要写一个函数来执行该算法，而该函数也可以调用其他函数处理它的一部分工作。

在实例程序中，你已经见过一些简单的算法，但还没有深入研究算法过程本身的特性。到目前为止，你所见过的大多数程序设计问题都太简单，马上就可以找到解决方法。然而，随着问题越来越复杂，需要周密地思考才能找到解决的策略，所以在写最终的程序前，需要考虑多个策略而不是一个策略。

本章将从解决几个经典数学中的问题开始，说明算法策略的具体形成过程，每个问题都可以用几种方法解决。通过用多种解决方案解决一个问题，可以学会如何比较不同的策略，以及如何选取最佳的策略。

6.1 测试素数

在西方，有记录可查的第一个数学家是约4000年前埃及的教堂中的一个牧师。从那时起，数学不断地发展，在古希腊进入了一个黄金时代，产生了许多数学家，如欧几里德、毕达哥拉斯、泰利斯、阿基米德以及希帕蒂娅。这些希腊数学家在许多不同的数学领域各有所长，但似乎都对研究非负整数的特性特别感兴趣，这就是现在被称为**数论**（number theory）的数学领域。

对希腊数学家和今天的数学家来说，数论中的一个重要问题是确定一个数是否为素数。如果一个正整数n只能被1和它本身整除，则这个正整数n就是**素数**（prime）。例如，23是一个素数，因为除了1和23以外，没有数能整除它。而35就不是素数，因为除了1和35之外，它还能被7和5整除。注意，按照这个定义，整数1不是素数，因为它只被1整除，这一点很重要。

第三，IsPrime会在有限步后结束吗？通过检查程序的结构就可以回答此问题。函数中仅有的长期运行的部分是一个for循环。从控制行可以看出，每次调用IsPrime函数，循环将执行n个周期。如果n非常大，函数会执行很长时间，但最终是会返回的。

因为满足这三个条件，所以图6-1中给出的函数IsPrime确实是一个算法。

6.1.3 说明IsPrime算法的正确性

除了确定该实现是一个算法外，确定算法是否正确也是很重要的。对每一个可能的n值，它都能给出正确的答案吗？很难用任何形式化的方法证明算法的正确性，进行这种证明所用的大多数技术超出了本书的范围。即使这样，对IsPrime的实现还是可以概述其证明的方法。

回想一下程序中的变量divisors的意义。在for循环的每个周期中，divisors记录了到这点为止遇到的约数个数。在for循环开始前，还没有发现任何约数，因此divisors被初始化为0。在循环的每一个周期中，当发现一个新的约数，程序将divisors加1。因此，在第i个周期结束时，divisors保存了1到i之间的约数个数。因为对每个循环值i，这个特性都为真，所以在第n个周期结束时也为真，于是变量divisors包含了1到n之间的约数个数。如果约数个数是2，则n一定是素数。

在循环开始时为真，在每一个循环周期结束前也为真的特性称为**循环不变式**(loop invariant)。对简单程序来说，通过说明维护了相应的循环不变式，可以容易地说明程序的正确性。对复杂的程序来说，证明正确性是相当困难的。在这些情况下，很少能依赖一些形式化的方法来确定程序是否确实递交了正确的答案。 188

通常，有两种方法可以增加你对程序正确性的把握。第一，按着程序代码一步一步执行，确信程序的行为和你的要求一致。这个过程被称为**桌面检查**（desk-checking）。学会用怀疑的眼光执行桌面检查并找到推理程序的技术需要实践和训练。第二种方法称为**测试**（testing），即利用尽可能多测试实例运行你的程序，对每种情况都检查执行结果是否正确。例如，可以写一个主程序，对1～1000之间的数调用IsPrime，然后检查输出，确信所有的数据都被正确分类，这样你对算法的正确性就会更有把握。

然而，完全严格的测试几乎是不可能的，记住这一点很重要。要测试的情况是不计其数的。当代著名的计算机科学家Edsger Dijkstra曾说过："测试能发现错误的存在，但永远不能发现它们不存在"。当你编写一个较大的复杂的程序时，必须把这个原理牢记在心。

常见错误 当你写一个程序时，尽量彻底地测试程序是很重要的。尽管这样，程序中还会有许多测试没有发现的错误，因为通常不可能将每一种可能出现的情况都测试一下。

6.1.4 改进算法的效率

和许多程序设计的问题一样，确定一个数是否为素数可以用许多种方法。图6-1所用的策略无疑有很多缺点。特别是该算法的效率比较低，如果你要检查的数字很大，它就不太可行了。例如，如果你对数字1 000 000调用IsPrime，该函数要检查1～1 000 000之间的一百万个数，确定它们是否为约数。测试所有的数是愚蠢的，因为1 000 000明显不是素数；显然，它能被2整除。可以确定的是，一定存在更好的方法，不用检查所有的数就能得到答案。

有多种修改这个基本的算法的方法，可以改善IsPrime函数实现的效率。例如，下面的三种修改方案都能明显改善IsPrime执行的效率：

1) 从1 000 000这个例子可知，IsPrime没有必要检查所有的约数。只要发现任何大于1

是无效的。

这种问题是很难发现的，它存在于for循环的控制行中：

```
for (i = 3; i<=sqrt(n); i+=2)
```

正如本章后面将介绍的，现代计算机能在相当短的时间内计算平方根。尽管如此，计算平方根还是比完成简单的算术运算（如乘和除）花的时间更长。如程序所示，IsPrime在每个for循环周期中都要调用sqrt函数，尽管每一次循环返回的结果都是相同的。因为n的值在循环内部没变过，sqrt(n)的值也不会变。为了避免一次又一次地调用sqrt，而得到的是同一个结果，可以在循环开始前先计算出sqrt(n)，并把它存入一个变量。例如，可以引入一个double类型的变量limit，用下列两个语句替换for循环的控制行：

```
limit = sqrt(n);
for (i = 3; i<= limit; i+=2)
```

这个简单的改变明显地改善了IsPrime实现的效率。

常见问题 *当在某一程序中使用循环时，检查一下是否在循环中存在某些放在循环开始前执行更好的计算。如果存在，则可以计算一次结果，把它存在一个变量里，然后在循环中使用这个变量，这样可以改善程序的效率。*

这个实现还有一个更难看出的问题。发现这个逻辑错误是很难的，因为它可能在测试中不出现。你可能用成千上万个输入值测试这个函数，每次得到的都是正确的答案。但某些人仍然可以用以前没有测试过的例子使函数失败。更令人烦恼的是，IsPrime的这个实现在一台机器上可能给出正确答案，而在另一台机器上却给出不正确的答案。

为了理解这个问题，可以回顾一下4.7.3节中的讨论。在这一节中，我们说过对浮点数判断严格的相等是很危险的。假设数n是某一个素数的平方。例如，假设n是49，它是7的平方。当对49调用sqrt函数时，它返回的值是什么？在严格的数学领域中，这个平方根是7，但计算机不是在这个领域中运作。浮点数仅仅是近似，sqrt(49)返回的可能是6.999999999999。尽管 191
这个数比7小一点点，但这个差别足以影响测试i<=limit的结果。如程序中所示，这个测试需要数学上完全精确，用浮点数不能保证这一点。如果i是7，而limit是6.999999999999，则循环的最后一个周期将不会被执行，程序永远不会检查n是否可以整除7。因为7是49唯一的因子，不检查7是否为49的因子意味着49将被错误地分类为素数。另一方面，如果sqrt(49)返回的是7.0或7.000000000001，IsPrime将会给出正确的答案。因此，这个实现的正确性取决于硬件如何执行浮点数运算。

让实现的正确性依赖于运行它的计算机的特性是一个严重的错误。这个问题很容易解决，可以修改这个程序使之与机器的精度无关。如果n的平方根小于某一界限，为了确保起见，函数总是多检查一个可能的约数。多测试一个约数没什么害处，这是确保答案正确性所付出的很小的代价。因此，要改正程序，所需要做的只是修改limit的赋值语

```
bool IsPrime(int n)
{
    int i, limit;

    if (n <= 1) return (FALSE);
    if (n == 2) return (TRUE);
    if (n % 2 == 0) return (FALSE);
    limit = sqrt(n) + 1;
    for (i = 3; i <= limit; i += 2) {
        if (n % i == 0) return (FALSE);
    }
    return (TRUE);
}
```

图6-3 IsPrime的正确实现

用这个策略计算GCD的函数如图6-4所示。

再重申一次，你必须问你自己几个有关实现的问题。GCD的brute-force的实现是不是总能给出正确的答案？它是不是总能结束，函数是否可能永远运行？

```
int GCD(int x, int y)
{
    int g;

    g = x;
    while (x % g != 0 || y % g != 0) {
        g--;
    }
    return (g);
}
```

图6-4 GCD的brute-force的实现

要知道程序是否总能给出正确的答案，需要看一下while循环中的条件

```
x % g != 0 || y % g != 0
```

while的条件指出在什么情况下循环将继续。为了找出什么样的条件会使循环终止，必须把while的条件取反。对一个包含&&或||的条件取反可以用德·摩根定理来完成，这个定理在第4章的4.3.2节已经介绍。德·摩根定理指出在while循环出口处，下列条件必须成立：

```
x % g == 0 && y % g == 0
```

从这个条件可以立即看出，g的最终值一定是公约数。要确定这事实上就是最大公约数，必须考虑嵌在while循环体中的策略。策略中的关键因素是该程序反向测试所有的可能性。GCD 194
永远不可能大于x（或y，对于这个问题），brute-force从这个值开始搜索。如果程序一旦出了while循环，它肯定已经测试了x和g的当前值之间的每一个值。因此，如果有一个较大的值能被x和y整除，那么程序在较早的while循环的迭代中就会发现这个数。

为了说明函数会终止，必须首先认识到即使没有找到最大公约数，g的值最终会到达1。此时，while循环肯定会终止，因为不管x和y有什么值，1总能被x和y整除。

6.2.2 欧几里德算法

然而，brute-force不是一个有效的策略。事实上，如果只关心效率的话，上一节中提到的GCD实现的brute-force算法是一个很差的选择。例如，考虑一下对1 000 005和1 000 000调用这个函数，会发生什么。brute-force在找到5之前将运行100万次while循环的循环体，其实这个答案只要考虑两个数就能很容易地确定了。

你所需要考虑的是找出一个算法，它比brute-force所需的执行步骤少，但却保证能结束，并返回正确的答案。这是聪明性和容易理解所付的代价。幸运的是，所需的具有创造性的想法已由古希腊的数学家欧几里得提出来了，他的*Elements*（book 7, proposition II）这本书中包含了一个解决这个问题的非常出色的算法。用自然语言，我们可以把欧几里德算法描述如下：

1) 取x除以y的余数，称余数为r。

2) 如果r是0，过程完成，答案是y。

3) 如果r非0，设x等于原来y的值，y等于r，重复整个过程。

你可以将这个算法翻译成GCD函数的实现，如图6-5所示。

```
int GCD(int x, int y)
{
    int r;

    while (TRUE) {
        r = x % y;
        if (r == 0) break;
        x = y;
        y = r;
    }
    return (y);
}
```

图6-5 GCD的欧几里德算法

GCD函数的这个实现同样能正确地找出两个数的最大公约数。它与brute-force的实现有两处不同。一方面，它能更快地计算出结果。另一方面，很难证明它的正确性。

在本章介绍的两个实例（检测素数和计算最大公约数）中，算法的选择对解决方案的效率有着极大的影响。在第17章中，我们还可以学到如何量化算法的效率。

6.3 数值算法

在现代世界中，计算机被应用到许多领域。结果，计算机程序员要编写许多不同类型的程序，程序中涉及各种不同的计算操作。有些程序员可能不用任何如平方根这样的数学操作，因此他可以根本不知道math库的内容。甚至对于那些使用平方根的程序员来说，大多数程序员也只满足于使用sqrt函数，而不考虑它的内部实现。然而，他们可以这样做的原因是某些程序员事先已经花时间和精力写了sqrt函数。计算机用来实现如sqrt这样的数学函数的技术称为**数值算法**（numerical algorithm），它是计算机科学的一个重要领域。 197

因为平方根在几何中频繁出现，所以数学家在2000多年的时间里依然对如何计算平方根的问题十分感兴趣，他们已经开发了解决这个问题的许多种算法。为了说明这些方案的技术（同时考虑数值算法的更一般的特性），在6.3.1节和6.3.2节中给出了计算平方根函数问题的两种不同的方法。

6.3.1 连续逼近

解决数值问题的最通用的策略之一是连续逼近的技术。**连续逼近**（successive approximation）是找出近似答案的一般策略，它由下列步骤组成：

1) 先对答案进行猜测。

2) 一旦有了猜测值，就可以用猜测值产生一个更佳的值。例如，假设你测试了你的答案，发现太大。你可以把它变得小一点，用这个小的值作为新的猜测值。相反，如果你原先猜测的值太小，那么可以选择一个较大的值作为下一个猜测值。

3) 如果你能保证，你的每一轮猜测与真实答案越来越接近，那么重复这个过程，最终将会产生一个足以满足应用需要的猜测值。

连续逼近技术的困难部分是第二步中的选择一个新的猜测值，使它满足第三步开始的条件。选择新的猜测值的策略必须考虑特定问题的一些知识。你所需要找到的是一系列的猜测值，每一个猜测值都比前一个猜测值更接近答案，随着序列的继续产生，可以使猜测值与答案达到任意接近程度。如果随着计算的项越来越多，一系列的值会接近于一个极限，那么这个数列被称为**收敛**（converge）的。

用连续逼近方法解决平方根问题的策略是由艾萨克·牛顿在17世纪发明的。计算平方根的牛顿法最好用一个实例来说明。假设要找出16的平方根。你可以随便做一个猜测。但必须小于16。你可以从8开始。你的答案是太大还是太小？如果8是平方根，那么8乘8应该是16。但8乘8是64，对16的平方根来讲，8太大了。你需要选择一个较小的数。但到底小多少？有没有使用这个问题的某些性质来选择下一个猜测值的方法？ 198

牛顿的观点是正确的平方根一定介于当前的猜测值和原先的值除以猜测值的结果之间。在这个实例中，猜测值是8，16除以8是2。也就是说，8太大了，2太小了。这两个值被认为是位于正确答案的两侧。为了保证更精确的猜测，可以取两个值的平均值。结果将更接近于16的平方根，虽然仍然是一个近似值。在这个实例中，将8和2的平均值5作为新的猜测值。

此时，只要简单地重复这个过程即可。16除以5是3.2。平均5和3.2产生一个新的猜测值4.1，它更接近于正确答案。如果你继续这个过程，后两个猜测值是4.001219512和4.00000018584。通过第四个猜测值，牛顿法已经产生了一个非常接近于正确答案的值。这个过程永远也不

6.3.2 报告错误

程序有很多种方法报告执行过程中出现的错误情况。本书中的程序用Error函数向用户报告错误，该函数在genlib库中定义。像printf一样，Error函数取一个控制字符串后跟另一个参数。当Error被调用时，显示一个字符串“Error:”，后跟一个控制字符串，像printf一样，格式代码的地方用一个值替换。在行尾，Error自动包括一个换行符，将光标移到下一行的开始。

Error和printf的主要区别在于Error不用返回到它的调用程序。一旦显示错误消息，整个程序就终止，就像它执行完主程序中的所有语句一样。因此，如果用-1调用Sqrt，屏幕上将会出现下面的输出：

199 ~ 200

```
Error: Sqrt called with negative argument -1
```

一旦出现错误消息，程序停止运行，不再执行任何语句。

在程序中响应一个错误的过程称为**错误处理**（error handling）。使一个程序输出一个错误消息并停止执行并不是特别复杂的错误处理策略。在如sqrt之类的库函数中，最好能提供一些调用程序可以用来采取正确行为的机制。但是，有关最好的错误处理策略的内容超出了本书的范围。当错误情况出现时，程序作出某些反应是非常重要的。如果不检查这些情况，程序可能给出不正确的答案，或根本不能产生答案。例如，如果Sqrt不检查负数参数，那么为了估计一个不存在的答案，程序将进入一个无限循环。使一个程序报告错误并终止是比使它永远运行更有用的响应。

6.4 级数展开

虽然牛顿法是计算平方根问题的一种可行的方法，但还有许多同样有效的方法。其中某些方法可以容易地应用在其他问题中。应用最广泛的方法之一是**级数展开**（series expansion），在这种方法中，函数的值通过对级数项的求和来估计。如果每加入一个新的项都会使总和更接近于一个所期望的值，则级数是收敛的，可以用此级数估计结果。

6.4.1 Zeno悖论

为了说明级数展开的思想，本节再一次探索希腊数学家发现的问题。在公元前5世纪，埃里亚的哲学家Zeno发明了悖论，他建议该行为是不可行的。假设你想穿过一个房间。要做到这一点，首先必须走到一半的地方。从这个地方开始，再走过余下的一半，这意味着再走过原始距离的四分之一。从这里开始，必须再走总距离的八分之一到下一个中点，一直这样走下去。因为余下的距离总是一个能被一分为二的量，所以Zeno认为，这个过程将永远不会结束，你永远不可能到达目的地。这个论点称为**Zeno悖论**（Zeno's paradox）。

201

在Zeno悖论中，横穿房间这个过程中的每一步都是走到剩余距离的一半。如果把Zeno问题以数学形式表示，那么已经走过的每段距离的总和可以表示为下式：

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}+\cdots$$

Zeno正确地观察到在这个级数中有无限个项[⊖]。但Zeno在公元前5世纪时无法认识到无限级数

⊖ 在一个数学的级数中，被加起来形成总和的每一个值称为项。由于级数中的项本身可以涉及一些数学运算（如Zeno计算中每个项所用的分数标记），所以级数中所用的项的定义与第2章表达式的讨论中所用的项的定义有所不同。

while循环的有效策略。如果用

```
while (sum != sum + term)
```

取代无限循环中的while控制行，那么只要term加到sum上时产生了一些可注意到的影响，那么循环将继续。

计算Zeno级数和的完整的主程序如图6-7所示。

```
main()
{
    double sum, term;

    sum = 0.0;
    term = 0.5;
    while (sum != sum + term) {
        sum += term;
        term /= 2;
    }
    printf("The sum of Zeno's series is %g\n", sum);
}
```

图6-7 Zeno级数和的主程序

6.4.2 用级数展开法设计平方根函数

和许多数学函数一样，平方根函数也有一个相关的幂级数，这样就可以用前一节介绍的求和技术产生近似结果。但是，找出使用哪一个幂级数需要熟悉微积分。一旦得到级数的公式，编写这样一个程序不费吹灰之力。

如果你不熟悉微积分，跳过下一节，从6.4.4节重新开始。如果你掌握了微积分，可以再来读这一节，看一下计算数学家们是如何提出用于实现数学库中的函数所需的技术。

6.4.3 估计平方根的泰勒级数展开（可选）

最有用的幂级数之一是泰勒（Taylor）级数，一般在大学一年级学习微积分时会遇到这个级数。18世纪的数学家布鲁克·泰勒发现，至少对一定范围之内的参数值，连续可导的函数可以用以下公式近似表示：

$$f(x) \cong f(a)+f'(a)(x-a)+f''(a)\frac{(x-a)^2}{2!}+f'''(a)\frac{(x-a)^3}{3!}+\cdots+f^{(n)}(a)\frac{(x-a)^n}{n!}$$

在这个公式中，符号a表示一个常量，记号f'、f''和f'''表示函数f的一阶、二阶和三阶导数，以此类推。这个公式称为**泰勒公式**（Taylor's formula）。

考虑f是平方根函数的情况。在数学中，取x的平方根与求x的1/2次幂是一样的，因此$f(x)$可以定义为

$$f(x)=x^{\frac{1}{2}}$$

为了找出f'，你需要做的就是用x^c求导的通用公式

$$cx^{c-1}$$

其中c为任意常数。在平方根函数中，c是1/2，因此一阶导数是

$$f'(x)=\frac{1}{2}x^{-\frac{1}{2}}$$

然后，运用同样的规则产生另外的一些导数。每一次都用前一次系数乘以前一次的指数，指

每一项与前一项有什么不同？为了回答这个问题，我们认为每一项都有三个独立的部分：

$$\texttt{corff}\frac{\texttt{xpower}}{\texttt{factorial}}$$

例如，对t_3来说，coeff是3/8，xpower是$(x-1)^3$，factorial是3!。然后，这个问题就变为研究这些部分是如何随项的变化而变化的。

例如，考虑xpower部分。将当前的xpower乘以$x-1$就得到了下一项的xpower。把当前的xpower值保存在一个变量中，就能确保在每个循环周期中通过一次乘法更新这个当前值。

factorial部分怎么办呢？对第i项，这个部分的值总是$i!$。阶乘有一个非常有用的特性：任何数i的阶乘是i乘以比它小的那个数的阶乘。这个特性可以很容易地用它的定义进行证明：

$$\begin{aligned}(i+1)!&=(i+1)\times i\times(i-1)\times(i-2)\times\cdots\times 2\times 1\\&=(i+1)\times i!\end{aligned}$$

因此，为了准备第（$i+1$）个循环周期，所需要做的就是把当前的factorial乘以$i+1$。

唯一剩下的部分是coeff。和其他几项一样，它也是由乘积产生的[⊖]。一般来讲，为了得到第（$i+1$）个循环周期的coeff值，只需要简单地把第i个周期的值乘以

$$\frac{1}{2}-i$$

206

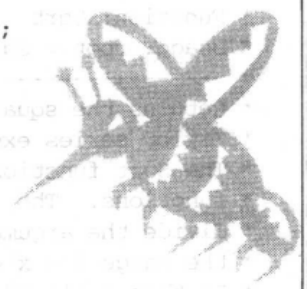
你可以用这些公式从每个部分的当前值中计算出下一项中各部分的值。如果变量coeff、xpower和factorial包含当前项的这些部分，那么为了计算下一项中这些部分的值，所需要做的只是将下列语句作为函数的主循环：

```
coeff *= (0.5 - i);
xpower *= (x - 1);
factorial *= (i + 1);
```

正如计算Zeno级数求和的程序一样，主循环必须执行到加入一个项的值不会改变总和的值为止。写一个循环测试这个条件，并初始化所有的变量，产生如图6-8所示的函数TSqrt。

```
double TSqrt(double x)
{
    double sum, factorial, coeff, term, xpower;
    int i;

    factorial = coeff = xpower = 1;
    sum = 0;
    term = 1;
    for (i = 0; sum != sum + term; i++) {
        sum += term;
        coeff *= (0.5 - i);
        xpower *= (x - 1);
        factorial *= (i + 1);
        term = coeff * xpower / factorial;
    }
    return (sum);
}
```



（如果$x\geqslant 2$，则函数失败）

图6-8 TSqrt的泰勒级数展开

⊖ 详细理解系数的逐项变化需要知道这些项是如何用泰勒公式计算的。如果你读了6.4.3节，就会知道系数是从平方根函数的导数中产生的。然而，即使你没有阅读这一节，你也可以顺利应用这个简单的公式。

```
    if (x == 0) return (0);
    if (x < 0) Error("Sqrt called with negative argument %g", x);
    correction = 1;
    while (x >= 2) {
        x /= 4;
        correction *= 2;
    }
    return (TSqrt(x) * correction);
}

/*
 * Function: TSqrt
 * Usage: root = TSqrt(x);
 * -----------------------
 * Returns the square root of x, calculated by expanding
 * the Taylor series around a = 1, as described in the
 * text.  The function is effective only if x is in the
 * range 0 < x < 2.  Term i in the summation has the form
 *
 *                    xpower
 *       coeff   *  ---------
 *                  factorial
 *
 * where coeff comes from ith derivative of the function,
 * factorial is i!, and xpower is the ith power of (x - a).
 * Each of these components is computed from its previous
 * value.
 */

double TSqrt(double x)
{
    double sum, factorial, coeff, term, xpower;
    int i;

    factorial = coeff = xpower = 1;
    sum = 0;
    term = 1;
    for (i = 0; sum != sum + term; i++) {
        sum += term;
        coeff *= (0.5 - i);
        xpower *= (x - 1);
        factorial *= (i + 1);
        term = coeff * xpower / factorial;
    }
    return (sum);
}
```

图 6-9 (续)

209

6.5 指定数值类型的大小

当你利用本章所示的这些数值应用时，控制数值数据的范围和精度是很重要的。在某种程度上，这些特性取决于硬件，因为计算机内的所有数据都存储在它的内存系统中。对不同类型的计算机，内存划分单元的大小可以不同。从程序员的观点来看，内部结构不同的影响主要体现在数值型的数据在不同的机器上可能受到不同的限制，这使得写一个可移植的程序非常困难。**可移植**（portable）程序是一个能在不同的计算机系统上以完全相同的方式成功运

6.5.2 无符号类型

在C语言中，整型类型int、long和short都可以在前面加上关键词unsigned。加上unsigned表示一种新的数据类型，这种类型仅允许表示非负值。由于无符号变量不需要表示负数，声明一个无符号类型的变量可以保存两倍的整数值。例如，int类型的最大数值是32 767，那么unsigned int类型的最大数值将是65 535。C语言允许将unsigned int类型缩写为unsigned，大多数使用这种类型的程序员都会遵从这个惯例。

Unsigned int类型的值在printf调用中用%u来显示。要显示unsigned long和unsigned short，可以分别用%lu和%hu。 211

在大多数情况下，不需要使用无符号类型，除非你需要对机器中的内部表示保持非常严格的控制。本书中介绍无符号数只是为了保持内容完整，事实上本书中没有使用这种类型。

6.5.3 浮点类型

和整数一样，浮点数也在一定长度的内存单元中表示，因此也有一定的大小限制。然而，对浮点数来讲，内存的限制影响了它的精度（可用的有效位有几位）以及范围。由于精度是硬件的函数，因此浮点数的精度随着机器的不同而不同。例如，在某台机器上，浮点数可以有10位有效位。在这台机器上，值1.0和1.00000000001是一样的，因为计算机不能精确表示10位之后的数。

与整型数一样，C语言提供了三种类型的浮点数。float是精度最小的，但比double类型占用的内存要少。然而，在近十年中，计算机内存越来越便宜，大多数C语言的程序员现在都习惯于将浮点数表示为更精确的double类型，本书也将遵从这个惯例。在C语言中，所有涉及float类型或double类型值的计算都把它当作double类型来处理；数据类型仅影响数据的存储。

然而，对一些精度要求非常高的计算，double类型的精度可能还不够。对于这些需要非常高的精度的应用，ANSI标准也包括了一种称为long double的类型，但这种类型在实际中很少使用。

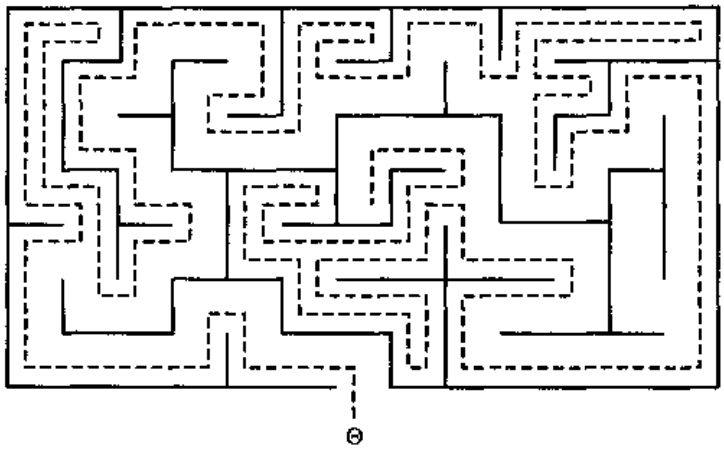
小结

在本章中，介绍了一些设计算法的通用策略。本章出现的实例都选自经典数学，因为数学提供了一些非常有趣的问题，除了数值之外，不需要掌握有关数据类型就能解决这类问题。当你在后面几章学过了其他的一些数据类型后，你会发现这里介绍的算法技术也可以应用于其他类型的问题。

本章介绍的要点包括：

- 解决特定问题通常有几种不同的算法。选择一个最适合应用的算法是程序员任务中最重要的部分。
- 当在不同的算法之间选择时，需要考虑许多不同的因素，包括效率、可维护性和清晰性。选择一个特定的算法需要在这些指标之间进行权衡。 212
- 在改善程序的其他方面时，决不能牺牲正确性。
- 证明程序的正确性是非常困难的任务。在大多数情况下，彻底的测试可以增加你对解决方案正确性的把握。然而，即使通过了良好的测试，还是会有一些隐蔽的错误。
- *brute-force*算法检查了每一种可能性，因而比较麻烦。这样的算法通常容易理解，但效率非常不高。

为了从迷宫中逃出，特修斯将他的右手放在墙上不离开，他走的路径如下图中的虚线所示。



如果你要写一个程序，为一个名叫特修斯的机器人实现右手法则。你可以访问一个包含下列函数的库：

```
void MoveForward(void);   /* 向前移动到下一个正方形区域 */
void TurnRight(void);     /* 右转但不移动 */
void TurnLeft(void);      /* 左转但不移动 */
bool IfFacingWall(void);  /* 如果特修斯面对墙，则返回TRUE */
bool IfOutside(void);     /* 如果特修斯逃离，则返回TRUE */
```

用这些函数写一个过程EscapeFromMaze，实现右手法则所提议的算法。

2. 在许多情况下，知道一个数是否为素数还不够，有时，还需要知道它的约数。每个大于1的正整数都能表示为素数的乘积。这个因数分解是唯一的，被称为**素数分解**（prime factorization）。例如，数字60可以分解为$2 \times 2 \times 3 \times 5$，其中每个约数都是素数。注意，同一个素数在因数分解中可以出现多次。

写一个程序显示数*n*的素数分解。下面是程序运行的一个实例：

```
Enter number to be factored: 60↵
2 * 2 * 3 * 5
```

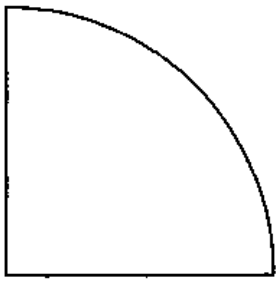
3. 希腊数学家对那些真约数的和等于该数本身的这些数特别感兴趣（*n*的真约数指的是小于*n*本身的任何约数）。这样的数被称为**完全数**（perfect number）。例如，6是一个完全数，因为它是1、2、3的和，1、2、3是小于6但能被6整除的数。类似地，28也是一个完全数，因为它是1、2、4、7、14的和。 215

写一个谓词函数IsPerfect，它取一个整数n，当n是完全数时，返回TRUE，否则返回FALSE。写一个主程序，用IsPerfect检查1～9999中的每一个数，看它们是不是完全数。当发现一个完全数时，将它显示在屏幕上。前两行的输出应该是6和28。此范围中的程序应该还能找到两个完全数。

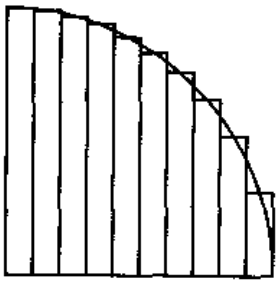
4. 当练习3中得到完全数的程序能够工作后，仔细考虑一下你用的算法。在不牺牲正确性的条件下设计程序的一个新版本，改善它的效率。

5. 修改本书中的牛顿算法，让它计算立方根而不是平方根。将该算法表示为一个函数CubeRoot，它取一个double类型的参数，返回另一个double类型的值，即参数的立方根。

9. 也可以用圆弧围住的面积的近似值计算π的近似值。考虑四分之一个圆：



它的半径r等于2英寸。利用圆面积的公式，很容易就能确定四分之一个圆的面积应该是π平方英寸。这个面积也可以通过将一连串的矩形面积加起来作为近似值，其中，每个矩形都有固定的宽度，其高度满足如下条件：圆弧通过矩形上面一条边的中点。例如，你把这个面积从左到右划分成10个矩形，可以得到下面的图： 217



矩形面积的和给出了一个四分之一圆面积的近似值。矩形越多，近似值越正确。

对每一个矩形来说，宽度w是一个常量，是半径除以矩形个数的结果。另一方面，高度h的值取决于矩形的位置。如果水平方向上矩形的中点位置是x，矩形的高度可用距离公式

$$h = \sqrt{r^2 - x^2}$$

来计算。矩形的面积可以简单地记为$h \times w$。

将它分成100个矩形，写一个程序计算四分之一圆面积。

10. 数学常量e可以用下列幂级数展开来计算：

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \cdots$$

写一个程序通过计算级数的和计算e的值，要求结果达到浮点数的精度限制。显示答案，小数点后取10位有效位。 218

第二部分

库和模块化开发

概要

如果某一天，你在软件行业找到了一份工作，你会发现很少有人完整地写完一个程序。在大多数公司里，程序员都是按团队工作，建立应用，这些应用的规模相当大，不是任何一个人可以单独完成的。使许多程序员能一起工作是现代软件工程最大的挑战。第二部分中的四章将介绍如何把大程序分解成独立的模块，如何设计这些模块，使它们能作为其他应用的库。

接口是库和它的使用者之间交换信息的媒介并给出了交换信息的结构。从概念上讲，程序设计的接口也表示有关边界特性的共识，提供了库的创建者和使用者之间所需的临界信息。 222

例如，考虑第5章引入的math库。math库中定义了一些函数，如sqrt。使用math库的程序员可以调用sqrt计算平方根，而不需要指出计算平方根的每个步骤。这些步骤是平方根函数实现的组成部分，它是由创建math库的程序员编写的。第6章介绍了两种可行的方法实现sqrt函数——牛顿法和泰勒级数展开。库的实现者正是使用这些方法中的某一种，或其他能计算出正确结果的算法。

掌握如何调用和实现sqrt是很重要的。然而，认识到调用函数和实现函数完全不相关也很关键。成功的程序员经常使用那些他们并不知道如何编写的函数。反过来，实现库函数的程序员也不能预见该函数的所有潜在应用。

为了强调实现库的程序员和使用库的程序员之间的区别，计算机科学家采用了特定的术语去谈及处于特定角色的程序员。很自然，实现库的程序员被称为**实现者**（implementor）。因为用户这个词通常指的是运行程序的人，而不是写程序的人，因此调用库函数的程序员被称为这个库的**客户**（client）[⊖]。

尽管客户和实现者对库有不同的看法，但他们都必须了解库设计中的某些问题。作为客户，不需要知道工作的细节，但必须知道如何调用它。而作为实现者，你不用关心客户程序员会如何使用库。但是，你必须给客户提供调用函数所需要的信息。对库中的每一个函数，客户必须知道下列内容：

- 函数名。
- 所需的参数以及参数的类型。
- 返回的结果类型。

这些信息正好是函数原型所提供的信息，这绝不是一个巧合。在C语言中，函数的原型和它的实现是分开的，因为它们传递信息给不同的读者。客户和实现者必须就函数的原型达成一致，这意味着这些信息是接口的一部分。相比之下，只有实现者是关心函数的实现。在接口中包含这个函数的原型，使客户可以使用该函数的过程称为**导出**（exporting）这个函数。 223

客户和实现者之间的关系可以用下图说明：

客户	接口	实现者
负责： • 如何使用函数	双方约定： • 函数原型	负责： • 函数是如何工作的

接口和头文件

在计算机科学中，接口是一个概念性的实体。它是由实现库的程序员和使用库的程序员之间的理解所组成，清楚地说明双方都需要的信息。然而，当你编写一个C语言的程序时，必须有一种方法将概念性的接口表示为实际程序的一部分。在C语言中，接口习惯上被表示为头文件。其实从第2章开始，你已经在使用头文件了，并且已经遇到了几个不同的头文件，包括stdio.h，math.h和genlib.h。每个头文件都指出了一个基本的库的接口。

接口的抽象概念和表示接口的实际的头文件之间的区别似乎很小。在许多方面，这个区别类似于算法和实现此算法的程序。算法是一个抽象的策略；而程序是一个算法的具体实现。

⊖ 在计算机科学中，术语客户有时指使用库的代码，有时指写这段代码的程序员。如果可能会混淆的话，我将用客户代码表示使用库的代码，虽然本书也将遵循标准，但只要意义明确，在这些情况下也可以使用客户这个词。

一对值来标识，通常记为 (x, y)，它指出了该点沿x轴和y轴的位置。这些值被称为点的**坐标**(coordinates)。坐标用相对于原点的英寸计量，原点的坐标为 (0，0)。从原点开始，x的值随着往右移而增加，y的值随着往上移而增加。

图形库中的坐标可以以两种形式出现：

- **绝对坐标** 用相对于原点的坐标值指定点在窗口中的位置。例如，图7-1中的实心点的绝对坐标是 (2.0, 1.5)。
- **相对坐标** 用该点与最后一次指定的位置在每根轴上的距离来指定某一位置。例如，图7-1上的空心点的绝对坐标为 (2.5, 1.5)。然而，如果用它相对于实心点的坐标来表示，这个点移动的相对坐标为 (0.5, 0.0)。如果要将这些点连成一条线，标准的方法是先用绝对坐标指定第一个点，然后用相对的方式指定连接线。

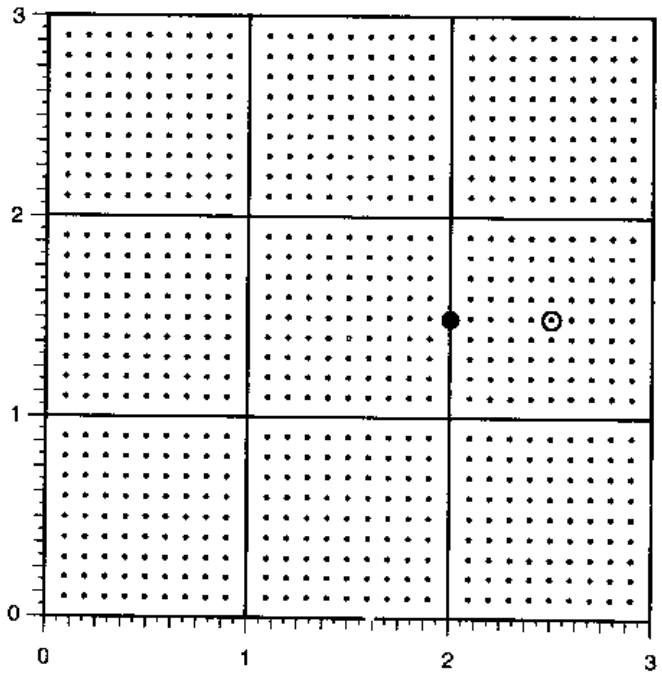


图7-1 图形库中的坐标

在绘图过程中，可以想象成在覆盖在屏幕上的透明图画纸上有一支笔。可以用绝对坐标把笔移到屏幕上的任何位置。一旦到了该位置，可以从当前位置开始，将笔连续在图形纸上移动 226
一个相对位置，画出一条直线。从这个位置开始，又可以画一条从上一个终点开始的直线[⊖]。

7.2.2 graphics.h接口的函数

graphics.h接口仅导出了很少的画图函数和过程。要画出复杂的图片，肯定需要更强大的功能，因为这个接口比较简单，所以我们能在一章中介绍一个完整的图形软件包。你也能很容易地理解它。该图形库包括下列函数：

⊖ 我的一个艺术家朋友告诉我，他们把下面的图案



看成是一条直线，因为它没有断。在本书中，我们用术语线表示一段直线。因此，这个图被认为是由三段头尾相连的线组成。

```
 *   DrawLine(dx, dy);
 *   DrawArc(r, start, sweep);
 *   width = GetWindowWidth();
 *   height = GetWindowHeight();
 *   x = GetCurrentX();
 *   y = GetCurrentY();
 */

/*
 * Function: InitGraphics
 * Usage: InitGraphics();
 * ----------------------
 * This procedure creates the graphics window on the screen.
 * The call to InitGraphics must precede any calls to other
 * functions in this package and must also precede any printf
 * output.  In most cases, the InitGraphics call is the first
 * statement in the function main.
 */

void InitGraphics(void);

/*
 * Function: MovePen
 * Usage: MovePen(x, y);
 * ---------------------
 * This procedure moves the current point to the position
 * (x, y), without drawing a line.  The model is that of
 * the pen being lifted off the graphics window surface and
 * then moved to its new position.
 */

void MovePen(double x, double y);

/*
 * Function: DrawLine
 * Usage: DrawLine(dx, dy);
 * ------------------------
 * This procedure draws a line extending from the current
 * point by moving the pen dx inches in the x direction
 * and dy inches in the y direction.  The final position
 * becomes the new current point.
 */

void DrawLine(double dx, double dy);

/*
 * Function: DrawArc
 * Usage: DrawArc(r, start, sweep);
 * --------------------------------
 * This procedure draws a circular arc, which always begins
 * at the current point.  The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle.  This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
```

图 7-2 (续)

```
#define_graphics_h
```

接口中的最后一行是

```
#endif
```

这些行的作用将在第8章中讨论，第8章主要介绍如何编写一个接口。现在你可以忽略这些行。虽然它们对编译器来说是很重要的，但对理解接口是如何工作的却毫无用处。

227 ~ 230

接口的其余部分仅由注释和函数原型组成。在剩余部分中，注释占头文件的90%。即使编译器忽略这些注释，但它们仍是接口中最重要的部分。接口实际的读者不是编译器，而是准备编写客户代码的程序员。注释的作用是帮助程序员全面理解这个抽象并使用接口提供的功能。

7.2.3 软件包初始化

graphics.h接口中的第一个过程是InitGraphics。正如接口中的注释所指出的，这个过程初始化图形库，必须在软件包中的其他函数和用printf在屏幕上显示任何输出以前调用它，库软件包需要一些初始化是很普遍的。当使用一个接口时，先把它通读一遍，看看是否需要初始化是一个比较好的策略。

7.2.4 画直线

函数MovePen和DrawLine是图形库提供的主要的画线工具。首先，我们从点 (0.5, 0.5) 向上画一条一英寸的直线。在任何使用图形库的主程序中，第一步总是

```
InitGraphics();
```

要画一条线，要先把笔移到点 (0.5, 0.5)：

```
MovePen (0.5, 0.5);
```

从这里开始，所有要做的就是画一条在*x*轴上不变，在*y*轴上向上移一英寸的线：

```
DrawLine (0.0, 1.0);
```

完整的程序如图7-3所示。

```
/*
 * File: oneline.c
 * ---------------
 * This program draws a single straight line.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

main()
{
    InitGraphics();
    MovePen(0.5, 0.5);
    DrawLine(0.0, 1.0);
}
```

图7-3 oneline.c

常见问题 确保任何使用图形库的程序的第一行要调用InitGraphics函数。作为一条规则，应该牢记库通常需要初始化。因此，应该检查每一个接口，看看是否需要进行初始化。

```
/*
 * Function: DrawArc
 * Usage: DrawArc(r, start, sweep);
 * --------------------------------
 * This procedure draws a circular arc, which always begins
 * at the current point.  The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle.  This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
 * position; if start is 90, the arc begins at the 12 o'clock
 * position; and so on.  The fraction of the circle drawn is
 * specified by the parameter sweep, which is also measured in
 * degrees.  If sweep is 360, DrawArc draws a complete circle;
 * if sweep is 90, it draws a quarter of a circle.  If the value
 * of sweep is positive, the arc is drawn counterclockwise from
 * the current point.  If sweep is negative, the arc is drawn
 * clockwise from the current point.  The current point at the
 * end of the DrawArc operation is the final position of the pen
 * along the arc.
 *
 * Examples:
 *   DrawArc(r, 0, 360)     Draws a circle to the left of the
 *                          current point.
 *   DrawArc(r, 90, 180)    Draws the left half of a semicircle
 *                          starting from the 12 o'clock position.
 *   DrawArc(r, 0, 90)      Draws a quarter circle from the 3
 *                          o'clock to the 12 o'clock position.
 *   DrawArc(r, 0, -90)     Draws a quarter circle from the 3
 *                          o'clock to the 6 o'clock position.
 *   DrawArc(r, -90, -90)   Draws a quarter circle from the 6
 *                          o'clock to the 9 o'clock position.
 */

void DrawArc(double r, double start, double sweep);
```

图7-4 DrawArc的接口项 233

例如，此文档建议可以调用

```
DrawArc (r, 0, 360);
```

在当前点的左边画一个完整的圆。

这条弧半径为r，从0度开始，即3点钟的位置。它延伸360度，因此创建了一个完整的圆。弧的开始位置是调用时笔的当前位置。相对于所画的圆的位置，这个位置在最右边，这个圆位于左边。

基于上述讨论，应该很容易编写一个画半径为半英寸，圆心在点（1, 1）的圆。所需要做的只是将笔移到圆最右边的起始点，然后调用DrawArc。主程序为

```
main()
{
   InitGraphics();
   MovePen(1.5, 1.0);
   DrawArc(0.5, 0, 360);
}
```

graphics.h接口的一部分并不成问题。因为C语言提供了自定义函数的功能，你可以自己创建这些工具。

7.3.1 定义DrawBox

为了说明这个过程，假设要定义一个过程DrawBox，它可以沿着坐标轴的方向画一个矩形框。写DrawBox的第一步是定义它的原型。这是设计的思考过程。你知道了过程的名字，但还必须考虑包含的参数。确定所需要的参数的最有用的策略是确定这个实现需要什么信息。你不能只给出一个等价于自然语言的命令："画一个矩形"。实现应该知道矩形有多大，把它放在屏幕的什么地方。获取这些信息的常用方法是由客户用参数的形式提供。

尽管如此，设计DrawBox过程还是有多种方法。一种可能的设计方法是仅用两个参数（width和height）指出方框的大小。为了指出方框的位置，应该调用MovePen。调用了MovePen后，就可以以此位置为参照来画方框。因此，在位置（x，y）画一个框，你应该写

```
MovePen(x, y);
DrawBox(width, height);
```

（这个实例不是最终的设计）

另一种DrawBox的设计是采用四个参数，x、y、width和height，它将设定位置和设定大小组合在一起，如下所示：

```
DrawBox (x, y, width, height);
```

因为第二种形式对调用程序来说更方便，通常都采用第二种形式，但两种设计方法都是可行的。

除了确定所需的参数个数外，还必须指明前两个参数的解释。在（x，y）位置画一个方框是什么意思？方框还没有一个明确的起始点。相对于矩形框，点（x，y）在哪里？对某些应用来说，比较方便的方法是使点（x，y）表示方框的中心。然而，更常用的策略是将方框的起始点定义在它的左下角，就如图形窗口的左下角是整个坐标系的原点一样。于是，点（x，y）指出了起始的位置。不管如何定义点（x，y）和方框的关系，需要做的主要工作是确保函数的文档使你的设计决策显得很清晰。

因此，DrawBox的一种可能的原型是

```
void DrawBox (double x, double y, double width, double height);
```

其中，x和y指出了方框的起始位置，width和height指出了它的大小。由于这个过程是自己创建的，而不是库的一部分，所以还需要定义它的实现。这个实现由按参数值画四条线所必需的步骤组成。在本章中，你已经画过一个方框，现在所需要做的只是将直接的坐标值转换为更一般的基于参数的形式：

236

```
void DrawBox(double x, double y, double width, double height)
{
    MovePen(x, y);
    DrawLine(0, height);
    DrawLine(width, 0);
    DrawLine(0, -height);
    DrawLine(-width, 0);
}
```

该实现先移到方框的原点，然后画形成该方框所需的四条线段。

现在你已经编写出这个过程，就可以改变在屏幕上画方框的程序的实现，使它用你的新工具。结果程序drawbox.c如图7-5所示。

7.3.2 定义DrawCenteredCircle

定义一个画完整的圆的函数是非常有用的。因为圆没有角，定义一个画相对于圆心的圆的函数DrawCenteredCircle是很有意义的。这个函数需要三个参数：圆心的x和y轴位置以及它的半径r。因此，DrawCenteredCircle的原型为

```
void DrawCenteredCircle (double x, double y, double r);
```

它的实现为

```
void DrawCenteredCircle(double x, double y, double r)
{
    MovePen(x + r, y);
    DrawArc(r, 0, 360);
}
```

虽然你也可以直接用DrawArc，但似乎DrawCenteredCircle更是裁剪得适合用户需求。首先，完整的圆在图形中相当普通，比一段弧出现的机会要多得多。另一方面，用高级函数可以使你从记住DrawArc如何解释角度的过程中解脱出来，而这些问题在画一个圆的时候是不需要考虑的。DrawCenteredCircle提供了方便性和简单性，这两点在程序设计中都有实用价值。

237 ~ 238

7.3.3 绝对坐标和相对坐标间的切换

MovePen过程用绝对坐标指出线的起点，然后用DrawLine通过相对坐标画线。对某些应用，将笔移到相对于原始位置的新位置，但不要画线也是很有用的。反过来，有时画一条到特定的绝对坐标位置的线也是很有用的。

函数GetCurrentX和GetCurrentY提供了写一个MovePen的相对版本和DrawLine的绝对版本的方便性。新的函数称为AdjustPen和DrawLineTo，它们的实现如图7-6所示。

```
/*
 * Function: AdjustPen
 * Usage: AdjustPen(dx, dy);
 * -------------------------
 * This procedure adjusts the current point by moving it
 * dx inches from its current x coordinate and dy inches
 * from its current y coordinate.  As with MovePen, no
 * line is actually drawn.
 */

void AdjustPen(double dx, double dy)
{
    MovePen(GetCurrentX() + dx, GetCurrentY() + dy);
}


/*
 * Function: DrawLineTo
 * Usage: DrawLineTo(x, y);
 * ------------------------
 * This function is like DrawLine, except that it uses the
 * absolute coordinates of the endpoint rather than the relative
 * displacement from the current point.
 */
```

图7-6 函数AdjustPen和DrawLineTo

```
#define   DoorWidth           0.4
#define   DoorHeight          0.7
#define   DoorknobRadius      0.03
#define   DoorknobInset       0.07

#define   PaneHeight          0.25
#define   PaneWidth           0.2

#define   FirstFloorWindows   0.3
#define   SecondFloorWindows  1.25
```

这些值都是实数，表示英寸，名字描述了它们在图中的物理含义。在程序中，用符号化的名字表示这些值使得改变它们的大小更方便，例如，在你要使房子更宽一些，窗子更大一些时。

7.4.1 使用逐步精化

现在已经可以开始实施了。如第5章中所讨论，处理大的程序设计问题的最好方法是使用逐步精化策略，将整个问题分解成一组较小的问题。为了将此策略应用到画房子这个问题，可以从最综合的层次开始：要画一栋房子。要给这个工作取一个名字，如DrawHouse，并定义它为一个过程。实现DrawHouse过程成为你的第一个子问题。为了完成这个实现，你可以将整个问题分解成较小的部分：画轮廓、门和窗户。这些工作是分解过程中的下一层子问题。执行这个策略直到所有的子问题都简化为能用已有工具解决的最简单的操作。

然而，和DrawBox过程一样，你还要确定DrawHouse过程是否需要参数。在上一节中，已将房子的大小定义为常数。还要指出房子的位置，因此DrawHouse过程似乎取x和y两个坐标值是合适的，这两个坐标值指出了房子在图形窗口中的位置。为了与DrawBox过程一致， 241
这两个值指出房子的左下角的位置。因此DrawHouse过程的原型应该是

```
void DrawHouse (double x, double y);
```

调用此过程将指示计算机画一栋房子，它的左下角位于点（x，y）。

一旦定义了原型，现在就能开始来完成主程序。你需要做的是指出房子应该在屏幕上出现的位置。例如，假设你要把房子画在图形窗口的中间。如在本章前面的7.2.6节所讨论的，你可以用函数GetWindowWidth和GetWindowHeight找出窗口中心的坐标。例如，如果声明了变量cx和cy，可以用

```
cx=GetWindowWidth()/2;
cy=GetWindowHeight()/2;
```

设定中心的坐标。然而，一旦定义了DrawHouse，图形本身是相对于左下角而不是中心的位置画的。我们如何把这两个位置关联起来？

你已经知道该房子的宽为HouseWidth。因此房子的左面的边离中心的位置是这个距离的一半。如果房子左边的边的坐标为

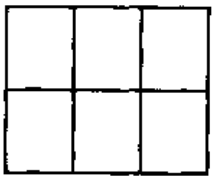
```
cx - HouseWidth/2
```

那么房子的中心是在屏幕的中心。你可以对y坐标重复同样的处理。唯一的差别在于房子的总高度是矩形和屋顶高度的和。因此房子左下角的y坐标为

```
cy - (HouseHeight + AtticHeight)/2
```

现在，你得到了房子左下角的坐标，就可以完成main的实现，程序如下：

为了设计适合于画窗的工具，尽可能考虑将这个问题通用化是不会错的。你的工具越通用，在各种情况下就越容易使用它们。保证你创建的工具能够广泛应用的一种方法是跳出当前问题的局限，而从一个更高的、更抽象的层次上认识所需要的操作。在房子这个问题上，右面的图描述了一个有几个窗格的窗。然而，当你集中于这个图本身时，你所见的是由两行组成的矩形网格，每行包括三个方格。如果有一个画一个矩形网格的过程DrawGrid，就可以用这个过程画每一组窗口。



DrawGrid需要什么参数？为了达到必要的通用性，必须保证DrawGrid过程不涉及房子问题的特性。因为用常量PaneWidth和PaneHeight会使过程只适用于本例房子的情况，最好是让调用程序将网格中的每一方框的宽度和高度作为参数来传递。调用程序知道这是画一扇窗，能为这个特定的应用提供PaneWidth和PaneHeight。这个过程本身只是画方框。除了网格中每个方框的高度和宽度外，DrawGrid也要知道完整的网格的坐标。为了与其他工具保持一致，DrawGrid将这些坐标解释为网格的左下角。最后，该过程必须知道网格中的列数和行数。因此DrawGrid的原型应该为：

```
Void DrawGrid(double x, double y, double width, double height,
              int columns, int rows);
```

假定已经有了函数DrawBox，DrawGrid的实现就非常简单。该实现由一对嵌套的for循环组成，它们在每一行中对每一列调用DrawBox，代码如下：

```
/*
 * Function: DrawGrid
 * Usage: DrawGrid(x, y, width, height, columns, rows);
 * ----------------------------------------------------
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid.  As always, (x, y) specifies the lower left corner
 * of the figure.
 */
void DrawGrid(double x, double y, double width, double height,
              int columns, int rows)
{
    int i, j;

    for (i = 0; i < columns; i++) {
        for (j = 0; j < rows; j++) {
            DrawBox(x + i * width, y + j * height,
                    width, height);
        }
    }
}
```

244

给定了DrawGrid的实现之后，可以通过用相应的参数调用DrawGrid来构造每扇窗口的模型。

尽管通常自顶向下设计问题是最好的方法，但自底向上实现问题通常也是最好的。先实现低层次的工具使它更易于调试程序的每个模块，这通常比一次性调试所有的程序更容易。这个策略称为**自底向上的实现**（bottom-up implementation）。

7.4.4 结束分解

给定了新的DrawGrid工具之后，画房子程序的剩下部分是一个逐步精化的最直接的例子。完整的程序house.c如图7-7所示。

```
                cy - (HouseHeight + AtticHeight) / 2);
}

/*
 * Function: DrawHouse
 * Usage: DrawHouse(x, y);
 * -----------------------
 * This function draws a house diagram with the lower left corner
 * at (x, y).  This level of the function merely divides up
 * the work.
 */

void DrawHouse(double x, double y)
{
    DrawOutline(x, y);
    DrawDoor(x + (HouseWidth - DoorWidth) / 2, y);
    DrawWindows(x, y);
}

/*
 * Function: DrawOutline
 * Usage: DrawOutline(x, y);
 * -------------------------
 * This function draws the outline for the house, using (x, y)
 * as the origin.  The outline consists of a box with a triangle
 * on top.
 */

void DrawOutline(double x, double y)
{
    DrawBox(x, y, HouseWidth, HouseHeight);
    DrawTriangle(x, y + HouseHeight, HouseWidth, AtticHeight);
}

/*
 * Function: DrawDoor
 * Usage: DrawDoor(x, y);
 * ----------------------
 * This function draws a door, with its doorknob.  As usual,
 * (x, y) specifies the lower left corner of the door.
 */

void DrawDoor(double x, double y)
{
    DrawBox(x, y, DoorWidth, DoorHeight);
    DrawCenteredCircle(x + DoorWidth - DoorknobInset,
                       y + DoorHeight / 2, DoorknobRadius);
}

/*
 * Function: DrawWindows
 * Usage: DrawWindows(x, y);
 * -------------------------
 * This function draws all the windows for the house,
 * taking advantage of the fact that the windows are all
 * arranged in two-dimensional grids of equal-sized panes.
```

246

图 7-7（续）

```
/*
 * Function: DrawCenteredCircle
 * Usage: DrawCenteredCircle(x, y, r);
 * -----------------------------------
 * This function draws a circle of radius r with its
 * center at (x, y).
 */

void DrawCenteredCircle(double x, double y, double r)
{
    MovePen(x + r, y);
    DrawArc(r, 0, 360);
}

/*
 * Function: DrawGrid
 * Usage: DrawGrid(x, y, width, height, columns, rows);
 * ----------------------------------------------------
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid.  As always, (x, y) specifies the lower left corner
 * of the figure.
 */

void DrawGrid(double x, double y, double width, double height,
              int columns, int rows)
{
    int i, j;

    for (i = 0; i < columns; i++) {
        for (j = 0; j < rows; j++) {
            DrawBox(x + i * width, y + j * height,
                    width, height);
        }
    }
}
```

249

图 7-7 (续)

小结

在本章中，我们开始研究接口的概念，接口是现代程序设计中最有用的想法。接口是库抽象的实现者和它的客户之间的连接点。接口指出了双方都需要知道的信息。在第8章，你将学到更多有关接口的知识，以及设计接口的策略。

本章介绍了一个特定的接口graphics.h，并将它作为介绍接口如何工作的通用实例。graphics.h接口使你可以在屏幕上的某一位置画线和弧等简单的图形。利用接口本身提供的函数，你也已经学会如何写一个其他的高级工具，如DrawBox、DrawCenteredCircle和DrawGrid，它们扩展了图形库的功能。

本章介绍的重点内容包括：

- 表示一个库的代码统称为*软件包*。软件包是某一抽象的程序表示，它是库的根本的概念基础。
- 库中的函数是由*实现者*编写，由*客户*调用。客户和实现者交汇的点称为*接口*。
- C语言中的接口用头文件表示。作为接口的头文件包含大量的文档以及由库导出的函数

```
DrawArc(4.0, -15, 2*15);
DrawArc(4.0, 180 - 15, 2 * 15);
```

16. 如何获取图形窗口中心的坐标?

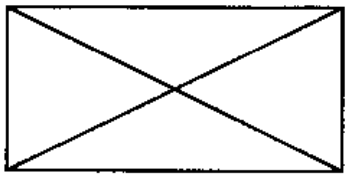
17. 建立类似于DrawBox和DrawCenteredCircle这样的新的过程有什么好处?

18. 当你设计一个作为工具的函数时，为什么跳出特定问题的范畴，更抽象地考虑该问题是非常有用的?

19. 自底向上的实现的意义是什么？优点是什么?

程序设计练习

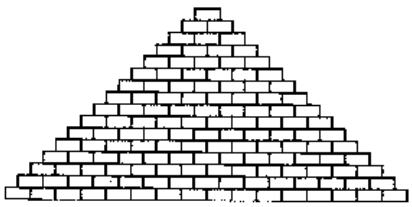
1. 写一个函数DrawCrossedBox，它的参数和DrawBox相同，但同时画上了矩形的对角线。例如，调用



```
DrawCrossedBox(0.5, 0.5, 1.0, 0.5);
```

将产生右面所示的图，该矩形的左下角在点(0.5, 0.5)。

2. 写一个程序，该程序画一个由一行行砖组成的金字塔，每向上移一层，行中砖的数目减1，如下图所示：



252

在你的实现中，应该用常量NBricksInBase指出最下面一行的砖数，用常量BrickWidth和BrickHeight指出砖的大小。

3. 画一个心型图案的一种方法是在一个旋转了45度后的正方形上面画两个半圆，如图7-8a所示。

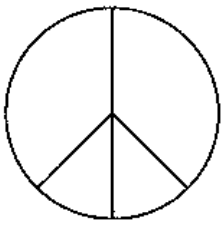
写一个程序，按这种结构在屏幕上画一颗心。该程序不要画出正方形上面的两条内部线，输出的图形看上去如图7-8b所示。



a) b)

图7-8 程序设计练习3的示意图

4. 1960年，下面这个符号



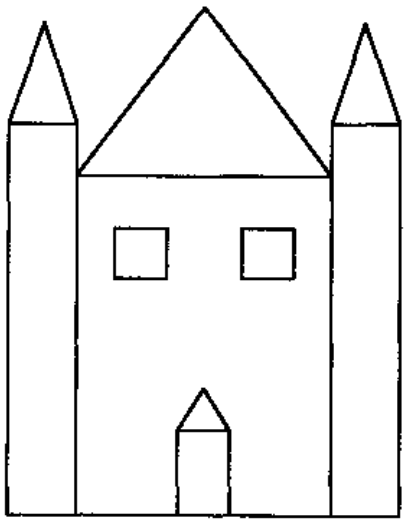
该能理解这些常量的意义。你的程序应该能做到修改这些常量会使图形有相应的变化。例如，如果将NTeethPerRow改为4，在新的图形中，每一行仅有四颗牙齿，但嘴仍然应该在水平位置的中间。南瓜脸的两个眼睛和嘴应该在两个方向上的中心到圆的边的中间位置上，因此改变HeadRadius也将改变这些特征的位置。表示南瓜的圆的中心应该是屏幕的中心。

8. 如果你要在练习7中设计的万圣节夜的南瓜旁画一栋房子，你可能要画一张Edgar Allen Poe描述的引座员的房子的图片，它的描述如下：

255

我看着前面的风景…寒冷的墙…空洞的像眼睛一样的窗…以及一个完全绝望的灵魂…

从Poe的描述中，你应该能想象出如右图所示的房子的样子。



写一个程序，画出如右图所示的房子，用下列常量指出各种尺寸：

```
#define HouseWidth          1.5
#define HouseHeight         2.0
#define HouseArch           1.0

#define TowerWidth          0.4
#define TowerHeight         2.3
#define TowerArch           0.6

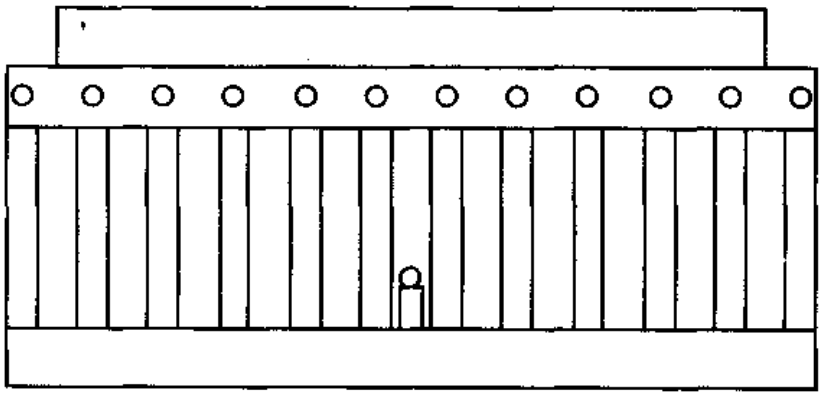
#define DoorWidth           0.3
#define DoorHeight          0.5
#define DoorArch            0.25

#define WindowLevel         1.4
#define WindowSize          0.3
```

以Arch结尾的常量指出矩形上面的三角形的高度。假设窗为正方形，因此只有一个尺寸WindowSize。

256

9. 写一个程序，画下面的位于华盛顿特区的林肯纪念馆的示意图：



和练习7和8一样，你的程序应该用下列常量定义图形的特征：

```
#define MemorialWidth       4.0
#define PedestalHeight      0.3

#define NumberOfColumns     12
#define ColumnWidth         0.15
#define ColumnHeight        1.0
```

第8章 设计接口：一个随机数库

And such a wall as I would have you think
That had in it a crannied hole, or chink
Through which the lovers, Pyramus and Thisbe
Did whisper often, very secretly

— Shakespeare, *A Midsummer Night's Dream*, 1595-1596

学习目标

- 了解评价接口设计的主要标准。
- 揭示程序如何利用伪随机数模拟随机行为。
- 理解库函数rand的行为。
- 学习如何使用算术运算修改伪随机数序列的范围。
- 认识接口项的常见类型。
- 学习写一个接口头文件的语法规则和习惯。
- 能够使用random.h接口提供的功能。

第7章已经介绍了接口的概念。此外，通过使用graphics.h接口，我们也了解了接口中有什么内容以及如何在程序设计中使用它们。但要完全理解接口，还必须学会如何实现它们。本章会让你设计一个新的接口以及它的基本实现。

按照所研究问题的范围，编写的接口可以是非常简单的，也可能是非常复杂的。如果仅考虑C语言的语法和结构，并没有许多新的规则要学。你已经学会了如何写注释和函数原型，它们是接口的主要组成部分。然而，一旦有了正确的算法，后面具有挑战性的工作并不是给这个接口编码，而是设计这个接口。重要的问题不是如何写一个接口，而是如何写一个好的接口。

设计一个好的接口是一个微妙的问题，需要平衡许多互相冲突的设计标准。本章将研究这些标准，说明它们的应用。为了使说明更加具体，本章将介绍一个库软件包开发的完整过程，该软件包提供了简单的随机数访问功能。

8.1 接口设计

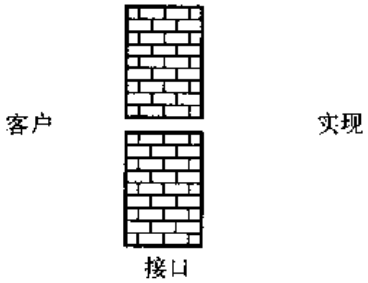
因为程序反映了它们所要解决的问题的复杂性，所以程序设计是非常困难的。只要你用计算机解决复杂性日益增长的问题，程序设计的过程也将变得更复杂。

在编写一个解决较大的或困难的问题的程序时，你必须处理大量的复杂性问题。其中有算法设计、要考虑的特殊情况、要满足用户的需求以及许多要保持正确的细节。要使得程序可管理，必须尽可能降低程序设计过程的复杂性。

在第5章中，我们已经学过使用函数和过程降低一些复杂性。接口提供了类似的作用，但它处于一个更高的细节层次上。函数让它的调用程序访问一组实现某一功能的一段程序。接口让它的客户访问一组实现某一程序设计中抽象行为的函数。接口对程序设计过程的简化程度在很大程度上依赖于它的设计。

为了设计一个有效的接口，必须在许多标准之间加以权衡。一般来讲，开发的接口应该

和实现之间互相通信。然而，墙的主要目的是保持两边分开。因为我们把它想象为库表示的抽象的边界，所以接口有时也被称为**抽象边界**（abstraction boundary）。理想情况下，库的实现中的所有的复杂性都位于墙的实现这一边。如果接口能对客户方屏蔽复杂性，那么接口就是成功的。把细节限制在实现范畴内称为**信息隐藏**（information hiding）。



信息隐藏的原理对接口的设计有重要的实际意义。当你写一个接口时，你应该确信没有显露出实现的细节，甚至在注释中也没有显露实现的细节。特别是，如果你同时写一个接口和一个实现，可能会不知不觉地在你的接口中记下你在写这个实现时的所有得意之处。一定不要这样做。写接口主要为了供客户使用，应该仅包含客户需要知道的内容。

同样，你应该在接口中设计这些函数，使它们尽可能简单。如果能减少参数个数或找到一种方法能消除特殊情况间的混乱，客户就容易理解如何使用这些函数。此外，限制接口输出 262
的函数个数通常是好的方法，这样客户不至于在大量的函数中迷失方向，找不到整体的意义。

8.1.3 满足客户的需要

任何事情都应该尽可能简单，而不是较简单。

——阿尔伯特·爱因斯坦

简单性只是接口设计原则的一个部分。抛弃接口中所有困难的或复杂的部分，就可以很容易地设计出一个接口。但这也使得一个接口变得毫无用处。有时，客户需要完成一些具有某种内在复杂性的任务。拒绝为客户提供他们所需的工具，可以使这个接口变得较简单，但这不是一个有效的策略。你的接口必须提供足够的功能以满足客户的需要。在接口设计中，学会在简单性和完备性之间达到合理的平衡是程序设计最基本的挑战之一。

在许多情况下，接口的客户不仅关心特定的函数是否可用，而且还关心实现的效率。例如，如果某一程序员正在开发一个航空控制系统，需要调用某一库接口提供的函数，这些函数必须迅速返回正确的结果。延迟返回的答案和错误的答案一样糟。

在大多数情况下，效率是实现而不是接口关心的因素。尽管如此，你还是经常会发现在设计接口本身时，考虑实现的策略是有价值的。例如，假设你有两个设计方案。如果能确定其中的一个能更容易更有效地实现，而另一个方案又没有绝对的竞争理由，选择这个设计是有意义的。

8.1.4 通用工具的优势

给我们工具，我们将完成工作。

——温斯顿·邱吉尔，广播演讲，1941

完全适合某一客户需要的接口对另外一些客户可能是毫无用处的。一个良好的库抽象要满足不同客户的需求。要做到这一点，它必须通用到足以解决很多问题，而不仅限于解决某种非常特殊的问题。通过选择一个能在客户使用这个抽象时具有足够灵活性的设计，就能创建一个应用范围很广的接口。

确保接口保持通用性有着重要的实用意义。当你写一个程序时，经常会发现需要一个特定的工具。如果你确定这个工具非常重要，值得放到一个库里，那么你需要改变你的思维方法。当你为库设计一个接口时，你必须忘记第一次使你想起要做这个工具的应用，而为大多 263

一些表示随机过程的机制，如抛硬币或掷骰子。模拟这种随机事件的程序称为**非确定性**（nondeterministic）程序。

8.2.2 随机数和伪随机数

因为早期的计算机主要用于数值应用，用计算机生成随机性的想法通常以生成特定范围内的**随机数**（random number）来表示。从理论上看，没有办法事先确定值的数是随机数，它将是一组等概率的值中的一个。例如，掷骰子生成1～6之间的随机值。如果骰子是公平的，那么没有办法预测将出现什么数。6个值出现的可能性是相等的。

虽然随机数的概念有直观的意义，但却是在计算机内部很难表示的概念。计算机根据一组内存中的指令序列工作，因此以一种确定的模式运行。如何遵循一组确定的规则产生一组不确定的结果？如果一个数是由一个确定的过程产生的，任何用户应该能够通过同一组规则工作并等待计算机的响应。 265

事实上，计算机确实能够用确定性的过程产生所需要的随机数。从理论上说，一旦用户能够遵循同一规则，并等待计算机的响应，那么这个策略就可以工作了，没有人会为此操心。在大多数实际应用中，这个数是否为真正随机是无所谓的；真正在乎的是这个数是随机出现的。对随机出现的数，它们的行为应该像统计学中的随机数，而且提前预测它们非常困难，但没有用户为此操心。由计算机内部的算法产生的“随机”数称为**伪随机数**（pseudo-random number），强调没有涉及真正的随机活动。

8.2.3 ANSI C中生成伪随机数

作为stdlib.h接口的一部分，ANSI C库包含了一个产生伪随机数的函数rand。在接口中给出的rand原型是

```
int rand(void);
```

它指出rand没有参数，并返回一个整数，这就是一个伪随机值，即每次调用rand返回的结果都不同。rand的结果保证是非负的，并且不大于常数RAND_MAX，该常数也是在stdlib.h接口中定义的。因此，每次调用rand时，它返回的是0～RAND_MAX的不同的整数，包括0和RAND_MAX。

RAND_MAX的值取决于计算机系统。在典型的Macintosh环境中，RAND_MAX的值是32 767。在典型的Unix工作站上，它的值是2 147 483 647。当你写一个用到随机数的程序时，不需要对RAND_MAX的值作精确的假设。相反，你的程序应该能在任意的RAND_MAX的系统上使用。如果你做到了这一点，你可以将一个在某一系统上工作的程序重新编译一下，它就能在另一个系统上工作。

运行图8-1中的程序randtest.c，将会显示rand的行为。

在我的一台计算机上，randtest.c产生了下列输出：

```
On this computer, RAND_MAX = 32767.
Here are the results of 10 calls to rand:
      16838
       5758
      10113
      17515
      31051
       5627
      23010
       7419
      16212
       4086
```

266

用这个想法可以开发出图8-2所示的cointest.c程序，它可以模拟抛硬币的过程。

```
/*
 * File: cointest.c
 * ----------------
 * This program simulates flipping a coin.
 */

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"

/*
 * Constants
 * ---------
 * NTrials -- Number of trials
 */

#define NTrials 10

/* Main program */

main()
{
    int i;

    for (i = 0; i < NTrials; i++) {
        if (rand() <= RAND_MAX / 2) {
            printf("Heads\n");
        } else {
            printf("Tails\n");
        }
    }
}
```

图8-2 cointest.c

该程序打印出字符串“Heads”或字符串“Tails”，每种输出的出现几率大约为50%。如果测试这个程序，可以得到如下的运行示例：

```
Tails
Heads
Heads
Tails
Tails
Heads
Tails
Heads
Heads
Heads
```

正面和反面有合理的组合，你不能很容易地找出它的模式。

在考虑如何将rand的结果转换为两种可能时，许多新的程序员最初可能对一些比较简单的方法感兴趣，如用取余数的运算。如果将rand的结果除以2，并取它的余数，结果是0或者

• 否则，如果产生的随机数小于最大值的3/6，返回3，以此类推。

问题是，你的代码没有完全实现这个想法。通过重复调用函数rand，在每一个if语句中都将产生一个新的随机数。该函数的结构依赖于每次产生的随机数都相同的假设。要了解这种情况，观察一下这个RollDie错误百出的实现，试着了解一下在什么条件下它将会返回2。为了使函数返回2，第一个if语句必须为FALSE，第二个语句必须为TRUE。第一个if语句的条件为FALSE的概率为5/6。第二个if语句会使得返回TRUE的概率为1/3，因为调用rand返回的是一个全新的随机数。在统计学上，两个独立事件一起出现的概率为两个独立事件出现的概率的乘积，因此RollDie返回2的概率是

$$\frac{5}{6}\times\frac{1}{3}=\frac{5}{18}$$

18次中出现5次的机会几乎是6次中出现1次的机会的2倍，这意味着RollDie函数返回2的机会比应该出现的机会大。为了解决这个问题，需要声明一个变量保存rand的结果，以后在每行中测试这个变量，如图8-4所示。

```
int RollDie(void)
{
    int r;

    r = rand();
    if (r < RAND_MAX / 6) {
        return (1);
    } else if (r < RAND_MAX * 2 / 6) {
        return (2);
    } else if (r < RAND_MAX * 3 / 6) {
        return (3);
    } else if (r < RAND_MAX * 4 / 6) {
        return (4);
    } else if (r < RAND_MAX * 5 / 6) {
        return (5);
    } else {
        return (6);
    }
}
```

(该实现仍然不正确)

图8-4 RollDie的第二次尝试

常见错误 当调用一个产生伪随机数的函数时，重要的是要记住每次调用这个函数都会产生一个不同值。如果要保存一个特定值，必须把函数的结果保存在一个变量中。

271

但是，这个实现还是错误百出。然而，第二个问题更加不明显了。在大多数系统上，RAND_MAX的值是按某种原因给出的。通常为RAND_MAX选择的值不一定是rand函数可能返回的最大值，而是系统中int类型的最大的正值。这个限制在所建议的RollDie的实现中引起了一个严重的问题，因为该程序中的某些中间结果可能大于最大的整数值。尽管

```
RAND_MAX * 2/6
```

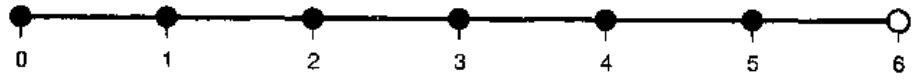
最终的结果适合int类型，但C语言中的优先规则指出RAND_MAX先乘2，再除6。产生一个所允许范围之外的整数称为**运算溢出**（arithmetic overflow）。如果出现这样的溢出，程序将不能产生预期的结果。你可以把表达式改为

严格小于1。因此，在这个过程结束时，可以得到一个至少为0，但严格小于1的随机实数。在数学中，可以等于一个端点值，但不能等于另一个端点值的实数范围称为**半开区间**（half-open interval）。在图中，不包括在范围内的端点用空心圆表示。因此变量d的可能的范围可以用下图表示：



273

下一步是换算这个值，使它覆盖正确的整型数的数目。例如，要模拟掷骰子，需要把这个值乘6，因此新的范围为



注意，在这个范围中有6个整数：整数0、1、2、3、4和5。6本身在可能值的范围外，因为d永远不可能为1。

在更一般的情况下，要将规范化的随机数乘以这个范围内的元素个数，元素个数可以用表达式

```
(high - low + 1)
```

表示。表达式中多出来的这个1是必须的，因为该范围是包括两个端点的。减法给出了两点之间的距离，它比闭区间中包含的整数少1。对掷骰子来讲，可能出现六个值是1、2、3、4、5和6，但6减1是5。为了计算出现的值的个数，需要将最大的数减去最小的数，然后加1。

下一步，向后截断该实数，使它成为一个整数。在C语言中，如果用强制类型转换将double转换为int，那么该转换是通过丢掉小数部分实现的。因此，如果要将一个已知的实型数转换为大于等于0，但严格小于6的数，将得到0、1、2、3、4和5中的某一个整数。

过程中的最后一步是将结果转换到所期望的范围内。已经有了整型结果的正确个数；仅有的问题是它们是从0开始。为了得到一组正确的可能性，只要简单地加上下界值即可。

可以把所有的这些步骤放在一起，写出如下的RandomInteger函数的实现：

```
int RandomInteger(int low, int high)
{
    int k;
    double d;

    d = (double) rand() / ((double) RAND_MAX + 1);
    k = (int) (d * (high - low + 1));
    return (low + k);
}
```

8.3 在库中保存工具

RandomInteger函数十分有用，因此可以把它放入某一个库中。这个过程的第一步是建立一个接口。一旦完成了这个接口，就可以在另一个文件中写出对应的实现。在大多数情况下，除了文件类型外，接口和它的实现的文件名是相同的。因此，如果接口被命名为random.h，一般会用random.c作为实现文件的名字。

274

8.3.1 接口的内容

接口的基本结构在第7章介绍的graphics.h实例中已作了说明。和本书中引入的所有接口一样，graphics.h主要由为使用该库的客户所编写的注释组成。这些注释是接口的关键

除了模板文件外，接口有时也需要用同样的#include行包含其他已经在程序中使用的接口。当用到这样的行时，所用的规则将在本章的8.4.3节讨论。

8.3.2 写random.h接口

如果将上一节讨论的规则用于编写random.h接口，你会认识到你的第一个任务是写初始的注释，这些注释将解释这个库提供了什么，谁可以使用它。在注释后面，应该包括接口的模板文件，本例中的模板文件是

```
#ifndef _random_h
#define _random_h
```

在random.h接口中，下一个要写的是有关RandomInteger过程的注释：

```
/*
 * Function: RandomInteger
 * Usage: n = RandomInteger(low, high);
 * ------------------------------------
 * This function returns a random integer in the range
 * low to high, inclusive.
 */
```

这段注释给客户提供使用这个函数所需的信息。例如，Usage这一行给出了一个调用函数的实例，对客户来说特别有帮助。这段注释也包括这个函数功能的说明，但没有说明函数是如何实现的。

接口的下一部分是函数本身的原型：

```
int RandomInteger(int low, int high);
```

这一行是接口中仅有的对编译器有实际意义的部分；其他都是注释或模板文件。

接口中的最后一行是简单的#endif行，它是接口的模板文件的一部分。 276

至今为止讨论过的random.h接口中的部分可以参见图8-5。如它的标题所示，图8-5中的接口仅是最初的版本。在本章后面，还会加入新的函数，以扩展它的功能。

```
/*
 * File: random.h
 * --------------
 * This file contains a preliminary version of a library
 * interface to produce pseudo-random numbers.
 */

#ifndef _random_h
#define _random_h

/*
 * Function: RandomInteger
 * Usage: n = RandomInteger(low, high);
 * ------------------------------------
 * This function returns a random integer in the range
 * low to high, inclusive.
 */

int RandomInteger(int low, int high);

#endif
```

图8-5 random.h的最初版本

8.3.4 构造客户程序

你可以写一个如图8-7所示的程序dicetest.c，来测试random.c的实现。主程序使你可以使用新的随机函数库，因此在dicetest.c文件中需要包括行

```
#include "random.h"
```

277
↓
278

使它能用RandomInteger函数。

```
/*
 * File: dicetest.c
 * ----------------
 * This program simulates rolling a die.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * ---------
 * NTrials -- Number of trials
 */

#define NTrials 10

/* Function prototypes */

int RollDie(void);

/* Main program */

main()
{
    int i;

    for (i = 0; i < NTrials; i++) {
        printf("%d\n", RollDie());
    }
}

/*
 * Function: RollDie
 * Usage: die = RollDie();
 * -----------------------
 * This function generates and returns a random integer in the
 * range 1 to 6, representing the roll of a six-sided die.
 */

int RollDie(void)
{
    return (RandomInteger(1, 6));
}
```

图8-7 dicetest.c

279

让我们迅速测试一下这个程序，确保它能工作。运行此程序的结果如下：

每次都返回同一序列值。

尽管rand的系统定义有利于调试，但一旦程序开始工作仍能够改变它的行为也是很重要的。然而，了解如何进行改变需要知道一些rand的实现的知识。

ANSI库通过记录最近生成的数产生伪随机数。每次调用rand时，它取最近生成的一个数，对这个数执行一系列的操作，产生下一个数。因为你不知道到底做了哪些运算，最好把整个操作看成一个黑盒子，旧的数从盒子的一边进入，新的伪随机数从另一边弹出。

在本章前面的8.2.3节中描述的randtest.c程序提供了rand内部操作的说明。在我的计算机上，前十次调用rand生成的数如下所示：

```
On this computer, RAND_MAX = 32767.
Here are the results of 10 calls to rand:
     16838
      5758
     10113
     17515
     31051
      5627
     23010
      7419
     16212
      4086
```

281

第一次调用rand产生16838。第二次调用对应于将16838放入表示内部实现的黑盒的一端，使5758从另一边弹出来，见右图。



同样，下一次调用rand时，该实现将5758放入黑盒子中，返回10113，见下图。在每次调用rand时，都重复同样的过程。黑盒内部的计算要满足以下条件：(1) 数值在合法的范围内均匀分布，(2) 该序列能持续很长时间才开始重复。



但第一次调用rand（返回16838的那次）的过程又是怎样呢？该实现必须有一个开始点。必须有一个整型数s被输入到黑盒子，以产生16838，见右图。这个用来使整个过程开始的初始值被称为随机数生成器的**种子**（seed）。每次程序开始时，ANSI的库实现将初始种子设为一个常量，因此总是产生同一序列值。将种子设为不同的值就能改变这个序列。要做到这一点，需要调用函数srand，它将一个新的种子作为参数。为了保证新的种子值在每次程序运行时都变化，标准的做法是用内部系统时钟值作为初始种子。因为时间一直在变，所以随机数序列也将变化。



可以调用函数time获取系统时钟的当前值，该函数在ANSI库接口time.h中定义，然后将返回值转换为整型数。这个技术允许你利用下列语句将伪随机数生成器初始化为某一个不可预测点：

```
srand((int) time(NULL));
```

虽然它只有一行，但基于系统时钟将随机种子设为一个不可预测值的操作是相对模糊的。如果这一行出现在客户程序中，客户必须理解随机数种子的概念、time函数以及神秘的常量

282

再定义一个函数以便提供所需的功能，与RandomInterger相对应，可以称为RandomReal。

8.4.1 产生随机实数

碰巧，作为RandomInteger实现的一部分，你已经使用rand函数来产生随机实数。生成随机整数过程的第一步就是生成一个0～1之间的随机浮点数。要实现RandomReal函数，一种方法是做同样的计算，返回结果。然而，这样的设计在某种程度上违反了保证库一致性的同一性原则。用这种方法，RandomReal没有参数，并返回一个处于预先设定范围内的浮点值。而RandomInteger函数的行为则不同。它有两个参数，并返回由输入定义的范围内的值。为保证一致性，最好让RandomReal有同样的设计。如果这样做的话，了解RandomInteger是如何工作的客户就能正确预测RandomReal的结构。因此，RandomReal应该有如下所示的原型：

```
double RandomReal(double low, double high);
```

它的实现基本上是由RandomInteger实现的前两行组成，只是换算因子是确切的两个端点之间的距离而不是包含在此范围内的整型数个数。因此，RandomReal的实现为

```
double RandomReal(double low, double high)
{
    double d;

    d = (double) rand() / ((double) RAND_MAX + 1);
    return (low + d * (high - low));
}
```

284

8.4.2 模拟一个概率事件

除了随机实数以外，在模拟随机行为的一般抽象中，还有另一种有用的随机变量的类型。假设你要写一个程序，其中需要一个事件以某种随机概率出现。例如，假设你的程序要模拟一条装配线，这条装配线平均每传送1000个零件就要出一次故障。用仿真的术语来讲，是每个零件有千分之一的故障率。在数学和统计学中，概率是0～1之间的数，因此，这个例子中故障率是0.001（1/1000）。

在这个例子中，结果只有两种可能：有故障或者没有故障。表示某个条件出现或不出现的只有两个结果的事实用布尔值表示是最恰当的。用TRUE表示检测到故障，它出现的概率是0.001。

在这种情况下，有一个以某种概率返回TRUE的谓词函数是很有用的。如果你已经用了这样的函数，它的名字可能为RandomChance，则可以用下列的代码表示这条装配线：

```
if (RandomChance(.001)) {
    printf("A defect has occurred.\n");
}
```

作为第二个实例，你也可以用RandomChance模拟投硬币：

```
if (RandomChance(.5)) {
    printf("heads\n");
} else {
    printf("tails\n");
}
```

与本章前面的8.2.4节中的实现方法相比，这种实现的优点在于它不需要客户理解rand函数本身的操作。客户仅依赖于更高级的random.h接口中定义的函数。rand函数的存在仅仅被认

子的结果，分解出下列几种情况：

- 掷出2、3或12，第一次掷骰子得到这些结果称为放弃，意味着你输了。
- 掷出7或11，第一次掷骰子得到这些结果称为自然赢，表示你赢了。
- 如果是其他数（4，5，6，8，9或10）。在这些情况下，掷到的数称为你的点，你继续掷骰子直到第二次掷到你的点或者是7，如果掷到你的点，你赢了；如果掷到7，你输了。如果掷到其他数（包括2，3，11和12，这些不再特殊处理），继续掷骰子直到出现你的点或7。

```
/*
 * File: craps.c
 * -------------
 * This program plays the dice game called craps.  For a discussion
 * of the rules of craps, please see the GiveInstructions function.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"
#include "simpio.h"
#include "strlib.h"

/* Function prototypes */

void GiveInstructions(void);
void PlayCrapsGame(void);
int RollTwoDice(void);
bool GetYesOrNo(string prompt);

/* Main program */

main()
{
    Randomize();
    if (GetYesOrNo("Would you like instructions? ")) {
        GiveInstructions();
    }
    while (TRUE) {
        PlayCrapsGame();
        if (!GetYesOrNo("Would you like to play again? ")) break;
    }
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * --------------------------
 * This function welcomes the player to the game and gives
 * instructions on the rules to craps.
 */

void GiveInstructions(void)
{
```

图8-8 craps.c

```
    printf("Rolling the dice . . .\n");
    d1 = RandomInteger(1, 6);
    d2 = RandomInteger(1, 6);
    total = d1 + d2;
    printf("You rolled %d and %d -- that's %d.\n", d1, d2, total);
    return (total);
}

/*
 * Function: GetYesOrNo
 * Usage: if (GetYesOrNo(prompt)) . . .
 * ------------------------------------
 * This function asks the user the question indicated by prompt
 * and waits for a yes/no response.  If the user answers "yes"
 * or "no", the program returns TRUE or FALSE accordingly.
 * If the user gives any other response, the program asks
 * the question again.
 */

bool GetYesOrNo(string prompt)
{
    string answer;

    while (TRUE) {
        printf("%s", prompt);
        answer = GetLine();
        if (StringEqual(answer, "yes")) return (TRUE);
        if (StringEqual(answer, "no")) return (FALSE);
        printf("Please answer yes or no.\n");
    }
}
```

图 8-8 (续)

287 ~ 289

程序本身是将自然语言描述的规则直接转换为C代码。当你通读craps.c程序时，应注意下列特征：

- 该程序包括接口random.h，因此，它能使用这个库中的函数。此外，这个程序仅应用那个库中的函数，从来没有直接调用rand（或srand、time）。随机数序列是调用Randomize来初始化的，每次掷骰子是调用RandomInteger产生的。
- 程序被分解成更小的单元，这些单元给出更详细的信息。这种分解有助于突出程序的结构，使你能理解这些单元是如何组合起来的。
- 掷两个骰子的问题出现在程序中的多个地方，因此模拟掷两个骰子、显示结果和记住总和这组动作被封装在一个函数RollTwoDice中，它也能被用在其他的地方。

小结

在本章中，你体验了编写一个接口和它对应的实现的过程。在某种层次上，你已经了解接口的语法结构和它包含的内容。也已经学习了一些接口设计的最一般的原则，即在解决一些大问题时非常重要一些原则。最后，还看到了在构造random.h接口时如何应用这些原则。

本章介绍的重点内容包括：

- 构造一个接口的挑战在于接口的设计而不是编码。
- 一个定义良好的接口必须具备同一性、简单性、充分性、通用性和稳定性。有时这些标

模拟掷两个骰子的过程？

11. 三个最常见的接口项是什么？

12. 如果定义一个名为magic.h的接口，接口的模板是什么样的？这些行的作用是什么？

13. **判断题**：每次程序运行时，rand函数一般生成的是同一个随机数序列。

14. 在随机数中，术语*种子*的含义是什么？

15. 调试包含随机数的程序，本章提供了哪些建议？

16. 在random.h接口的最终版本中，定义了哪些函数？这些函数在哪些场合下使用？

程序设计练习

1. 在你的计算机系统上运行randtest.c程序，计算机上的RAND_MAX的值是什么？

2. 写一个显示2～100之间的偶随机数的程序。

3. 写一个随机显示7位电话号码的程序。输出应该服从在美国应用的以下规则：

- 输出的3～4位之间有一个破折号，如555-1968。
- 前两位都不能是0或1。这条规则实际上在美国的许多地方都已不采用，不过在这个问题中你还应该使用这条规则。因此，你的程序可能产生781-9902这样的电话号码，而不会产生718-9902这样的电话号码。 292

4. 写一个程序显示从52张扑克牌中随机抽取到的一张牌的名字。每张牌有它的等级（A, 2, 3, 4, 5, 6, 7, 8, 9, J, Q, K）和花色（梅花，方片，红桃，黑桃）。程序应显示完整的牌的名字，如下列运行示例所示：

```
Queen of Spades
```

5. **Heads. . . .**
Heads. . . .
Heads. . . .
A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.

—*Rosencrantz and Guildenstern Are Dead*, Tom Stoppard, 1967

写一个程序模拟反复投硬币，直到连续出现三个正面为止的情况。此时，你的程序应该显示投掷硬币的总次数。下面是程序可能的运行结果：

```
tails
heads
heads
tails
tails
heads
tails
heads
heads
heads
It took 10 flips to get heads 3 consecutive times.
```

```
 2        2464
 3        1207
 4         627
 5         311
 6         166
 7          89
 8          40
 9          21
10           8
11           4
12           1
13           0
```

正如这些数字所指出的，每年在样本中衰减了约一半的原子。在物理中，表示这个观察结果的常用说法是这个样本每年有一个半衰期。

295

8. 随着计算机在学校中越来越普及，找到如何在教学过程中用好计算机的方法是很重要的。这促进了教学软件业的发展，而且生产了许多帮助教学的工具。

作为教学应用的一个实例，写一个程序，它能产生一系列简单的算术问题让学生回答，如下所示：

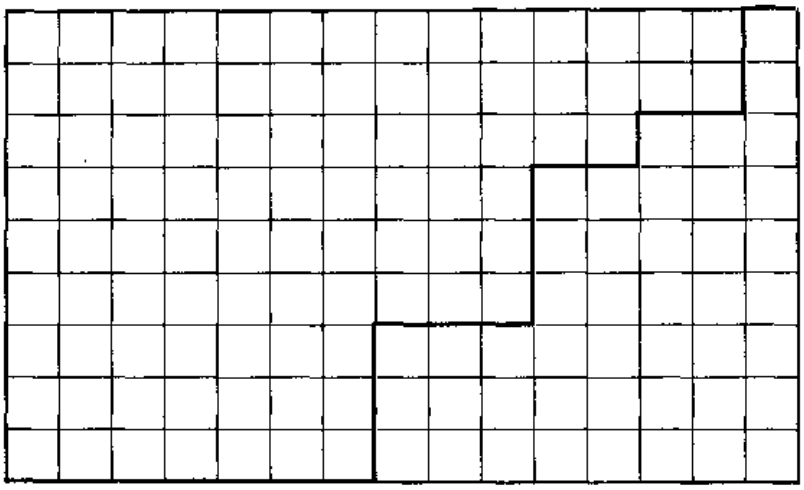
```
Welcome to Math Quiz!
What is 14 + 2? 16↵
That's the answer!
What is 17 - 15? 17↵
That's incorrect.  Try a different answer: 15↵
That's incorrect.  Try a different answer: 3↵
No, the answer is 2.
What is 20 - 16? 4↵
That's the answer!
What is 9 + 4? 11↵
That's incorrect.  Try a different answer: 13↵
That's the answer!
What is 11 - 1? 10↵
That's the answer!
```

你的程序应该满足以下需求：

- 它可以连续问五个问题。因为有这样的限制，所以问题数应该用#define常量定义，使之容易修改。
- 每个问题应该是仅涉及两个数的简单的加减问题，如“2＋3等于多少？”或“11－7等于多少？”。每一个问题的类型（即加或减）应该是随机选择的。
- 为了保证问题适合一、二年级的学生，任何数字（包括答案）都不能小于1或大于20。这个限制意味着程序不应该问类似于“11＋13是多少？”或“4－7是多少？”的问题，因为答案超出了合法的范围。在这些约束下，你的程序应该随机选择数字。
- 对每个问题，程序应该给学生三次回答的机会。如果学生给出了正确的答案，程序应该能以某种方式表示祝贺，并继续下一问题。如果学生三次都没能给出正确的回答，程序应该给出答案，并继续另一个问题。

296

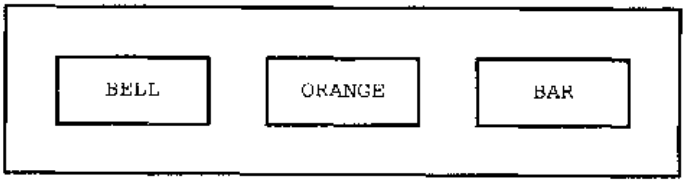
9. 尽管练习8中的程序在学生正确回答问题后会给出鼓励。但是只反复显示如“对了！”



写一个程序，用图形库描绘出通过城市的一条随机路线。开始时，将笔移到右上角。从这里开始，以随机的方式选择方向，画一条水平的或垂直的线到下一个交叉点。继续这个过程，直到你到达家中，但确保不要走到地图的外面去。 298

该程序不需要画一张完整的地图，只要直接显示这条路线就足够了。然而，如果想要更熟练地使用图形库，应该画出例子中所示的随机路线的完整的图。为了用粗线条表示随机路线，可以在实际格子线的每一边画一根直线。

12. 有一种叫做“碰运气”的游戏工具[⊖]。它有三个轮子，在一个很窄的窗口后面旋转。每个轮子上有下列符号标记：CHERRY、LEMON、ORANGE、PLUM、BELL和BAR。然而，在某一时刻你只能看到每个轮子上的一个符号。例如，窗口可能显示以下的配置：



开始游戏时，轮子开始旋转，最终出现某一个新的配置。如果该配置与规则规定的某个赢的组合相匹配，你就赢了。否则，你就输了，会被扣一分。下表显示了一个典型的赢模式的集合以及相应的得分：

BAR	BAR	BAR	得分	250
BELL	BELL	BELL/BAR	得分	20
PLUM	PLUM	PLUM/BAR	得分	14
ORANGE	ORANGE	ORANGE/BAR	得分	10
CHERRY	CHERRY	CHERRY	得分	7
CHERRY	CHERRY	—	得分	5
CHERRY	—	—	得分	2

符号BELL/BAR指的是在这个位置出现BELL或BAR，破折号指的是这里可以出现任何符号。因此，在第一个位置是CHERRY，不管其他轮子出现了什么，自动得到2分。注意，符号LEMON不会有得分，甚至碰巧三个都是LEMON时也一样。

写一个程序，模拟这个游戏。该程序给用户50分的初始分，让用户玩，直到分被扣完或

⊖ 本题在英文原书基础上有所改编。——编辑注

第9章 字符串和字符

Surely you don't think numbers are as important as words.

— King Azaz to the Mathemagician, *The Phantom Tollbooth*, Norton Juster, 1961.

学习目标

- 理解枚举的原理和用整数编码作为定义新的数据类型的策略。
- 能定义和操作C语言中的枚举类型。
- 理解计算机系统如何表示数据类型char，如何操作char类型的对象。
- 学习如何使用ctype.h接口中的函数。
- 理解抽象类型的概念。
- 能用strlib.h接口中的函数实现字符串操作。

301

到目前为止，你在本书中看见的绝大多数的程序设计的例子都是用数字作为基本的数据类型。正如Juster在Mathemagician中所坚持的，数字确实很重要，但在世界上还有许多其他类型的数据。目前，计算机较少用于数值型的数据，而较多用于**文本数据**（text data），即由键盘和屏幕上出现的独立的字符组成的信息。现代计算机处理文本数据的能力推动了字处理系统、在线参考库、电子邮件以及其他许多非常有用的应用的发展。

文本数据的概念在第2章中已正式引入，以本书的第一行代码为例，它包括字符串

```
"Hello, world.\n"
```

从那时开始，你已经见过如何用GetLine读一个字符串的值，如何用printf显示字符串，如何用StringEqual比较两个字符串。然而，这些操作仅仅是字符串操作中的很少的一部分。为了发挥文本数据的所有能力，需要知道如何以更复杂的方法操作字符串。本章中，你将学习如何使用字符串操作的库，它将帮助你写出更有创造性、更有趣的包含文本数据的程序。

因为字符串是由独立的字符组成，因此了解字符是如何工作的，以及它们如何在计算机内部表示是非常重要的。因此，本章重点介绍数据类型char以及数据类型string。然而，在研究这些类型的细节之前，需要从更一般的观点了解数据的表示。

9.1 枚举的原理

随着计算机技术的发展，越来越多的信息要以电子化的方式存储。为了将信息存储在计算机中，需要将数据表示为机器能使用的形式。特定内容的表示取决于它的数据类型。整数在计算机内有一种表示方式，而浮点数有另一种表示方式。尽管你不知道这些数据究竟是怎样表示的，但是这取决于计算机能在它的内部存储器中存储数据这个事实。除了数字之外，还有许多有用的数据类型，因此计算机必须也能表示这些非数值型的数据。

为了洞察非数值型数据的特性，考虑一下你自己在一年中提供给学校和机关的信息。例如，如果你生活在美国，你要将你的年退税数据提供给国内税收服务部。这些信息绝大部分是数值型的，如你的工资、扣除额、税、代扣所得税等。有些是由文本数据组成的，如你的名字、地址、职务。但还有一些项目不能简单归入这两类。例如，问题之一是

302

```
#define Tuesday      2
#define Wednesday    3
#define Thursday     4
#define Friday       5
#define Saturday     6
```

为了用这种方法表示枚举类型，必须显式指定整数编码。如果要引入一个变量保存这种类型的值，可以把它定义为int型。

也可以用这种方法表示退税单上的报税身份，可以定义如下常量：

```
#define Single                          1
#define MarriedFilingJointReturn        2
#define MarriedFilingSeparateReturn     3
#define HeadOfHousehold                 4
#define QualifyingSurvivingSpouse       5
```

最后一个常量的名字已经从税单上的名字改为符合C语言规则的名字，因为C语言的名字中不允许出现圆括号。

一旦定义了这些常量，就可以声明一个int类型的变量来表示报税身份：

```
int filingStatus;
```

9.1.3 定义新的枚举类型

尽管上一节中描述的方法在C程序中很常见，但它并没有完全发挥该语言提供的功能的所有优势。如果不使用int类型表示非数字值，在C语言中，可以定义一个真正的类型名去表示一种枚举类型。在机器内部，这两种策略产生的结果完全相同：枚举类型中的每个元素都用一个整数代码表示。从程序员的观点来看，定义单独的枚举类型有三个优势： 304

- 编译器能选择整数代码，从而将程序员解脱出来。
- 有一个独立的类型名使程序容易读，因为声明时可以使用有意义的名字，而不是通用的符号int。
- 在许多计算机系统中，使用显式定义的枚举类型的程序容易调试，因为编译器可以为调试系统提供有关该类型行为的额外信息。

C语言提供了多种定义新的枚举类型的语法形式。在本书中，所有新的枚举类型都是用如下所示的语法定义的。该定义以关键词typedef开头，它表示引入了一种新的类型名。在这里，关键词typedef后面紧跟着关键词enum，它指出新的类型是枚举类型。然后，枚举类型中的每个元素的名字列在一对花括号中，再接下去是类型的名字和一个分号。（关键词typedef也可以用在其他类型定义中，这将在第16章讨论。）

语法：枚举类型的定义

```
typedef enum {
    list of elements
} type name;
```

其中：

*list of elements*是由组成枚举类型的每个值的名字组成的。表中的元素用逗号分开。每个元素后也可以紧跟一个等号和一个整数常量，它指出了特定的内部表示。

*type name*指出新的枚举类型的名字。

定义指南针上的四个基本方向。每个这样的定义都引入了一个新的类型名和对应于该类型元素的一组常量。

事实上，genlib.h中的bool类型的定义就是一个枚举类型：[⊖]

```
typedef enum {FALSE, TRUE} bool;
```

9.1.4 枚举类型的操作

一旦在表达式中用到枚举类型的值时，C编译器自动将它转换为整数。因为每个枚举常量都用一个整数值表示，所有涉及枚举类型的计算只是用到它的整型代码代替。例如，如果你声明一个变量weekday为weekdayT类型，你仍然可以用下面的表达式

```
weekday = (weekday + 1) % 7;
```

就和在calendar.c程序中一样。变量weekday可以是0～6之间的某一个内部整数值。该语句将此整数值加1，然后通过模7取余操作保证结果还在0～6之间。枚举类型的所有算术运算都与整数一样。

然而，在C语言中，使用枚举类型时要非常小心，因为编译器不检查计算的结果是否是某一枚举类型的合法成员。例如，如果你误写了一个语句

```
weekday = weekday + 1;
```

如果weekday的值碰巧为Saturday，就会遇到麻烦。计算机将取表示Saturday的值6，对它加1，把这个值7存回到变量weekday，尽管7不是weekdayT类型的合法元素。C编译器应该生成一段额外的代码去检查这个条件，并报告错误，但很少有编译器这样做。当然，如果你将weekday作为一个整数，不考虑取余数的操作也是会产生错误的，因此，选择使用枚举类型不是问题的起因。

9.1.5 标量类型

行为类似于整数的类型，如枚举类型，称为**标量类型**（scalar type）。在C语言中，当你在表达式中使用标量类型时，它们被自动转换为整数。此外，标量类型可以用在任何整数类型出现的地方。例如，一个枚举类型的变量能被用在switch语句的控制表达式中。假设directionT被定义为 307

```
typedef enum {North, East, South, West} directionT;
```

下面的函数返回参数指定的方向的反方向：

```
directionT OppositeDirection(directionT dir)
{
    switch (dir) {
      case North: return (South);
      case East:  return (West);
      case South: return (North);
      case West:  return (East);
      default:    Error("Illegal direction value");
    }
}
```

⊖ 在一个表示布尔条件的C语言表达式中，整数0表示条件为假，尽管整数1是常量TRUE的“官方”值，但实际上任何非0的值都被解释为真。C程序员经常利用C语言解释非0整数为真的条件，尽管本书中的程序没有这样做。限制你自己在条件中使用明确的布尔值，将有利于你逐渐养成良好的程序设计习惯，这将使得你的程序比较容易读和调试。

即65。以同样的方法，能找出任何字符的编码。然而，在大多数情况下，并不需要这样做。虽然知道字符在计算机内部用什么数字编码很重要，但知道哪个数字值对应于哪个特定字符并不是很有用。当你键入字母A时，键盘中的硬件自动将此字符翻译成ASCII代码65，然后把它发送给计算机。同样，当计算机把ASCII代码65发送到屏幕时，屏幕上会出现字母A。

表9-1 ASCII代码

	0	1	2	3	4	5	6	7	8	9
0	\000	\001	\002	\003	\004	\005	\006	\a	\b	\t
10	\n	\v	\f	\r	\016	\017	\020	\021	\022	\023
20	\024	\025	\026	\027	\030	\031	\032	\033	\034	\035
30	\036	\037	空格	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	\|	}	~	\177		

9.2.3 字符常量

当你在一个C语言程序中用到某一特定字符时，标准的方法是指定一个**字符常量**（character constant），即将所需的字符括在一对单引号中。例如，要指出字母A的ASCII值，所需要做的只是写'A'。C编译器知道这个符号指的是用字母A的ASCII值，即65。同样，可以用' '指出空格，或用'9'指出数字9。注意，常量'9'指的是一个字符，不应该与整数值9混淆。作为一个整数，值'9'是ASCII表中给出的这个字符的值，即57。

只要你的计算机使用ASCII字符集，就应该用整数65代替字符常量'A'。程序也应该完全以这种方式工作，但这样程序将会很难读。你要牢记，会有其他程序员要读你写的程序。除非这些程序员能记住ASCII表，否则看见程序中所写的整数65不会立刻联想到字母A。而字符常量'A'则可以更直接地传递出这个意义。

本书用表9-1给出了字符如何在机器内部表示的更具体的理解。一旦你心中有了这样一个概念，你应该忘记特定的字符代码，而集中于字符本身。

常见问题 避免在程序中用整数常量表示ASCII字符。所有的字符常量应该用单引号将此常量括起来表示，如'A'或'*'。

9.2.4 ASCII代码方案的重要特性

尽管考虑特定字符的代码不是很重要的，但以下两个ASCII表的结构特性是需要牢记的：

309～310

1) 表示数字0～9的字符的代码是连续的。尽管你不需要知道哪个代码对应数字字符'0'，但要知道数字'1'的代码是比'0'的代码大1的整数。同样，如果给'0'的代码加9，就是字符'9'的代码。

2) 字母按字母顺序分成两段：一段是大写字母（A～Z），另一段是小写字母（a～z）。然而，在每一段中，ASCII值是连续的，因此，为了得到它们的ASCII代码，你可以一个个字母

```
"Bother," said Pooh.
```

ASCII表中的许多特殊字符都没有明确的名字，在程序中使用它们的内部代码来表示。在此过程中，唯一的不便之处是这些特殊字符的数字代码是用八进制表示的，通常称为**八进制计数法**（octal notation）。在八进制计数法中，每一位数字是它右边数字的八倍。例如，字符常量'\177'表示ASCII值为八进制数177的字符（这个字符对应于表示Delete键的代码，在某些键盘上标记为Rubout）。在数值上，八进制值177对应于整数

$$1 \times 64+7 \times 8+7$$

结果为十进制数127。

ASCII代码系统中的许多特殊字符实际上很少使用。对大多数程序设计应用来说，只需要知道换行（'\n'）、tab（'\t'）和空字符（'\0'）就可以了，这些内容将在第14章讨论。

9.2.6 字符运算

在C语言中，字符值能像整数一样计算，不需要特别转换。结果是根据其内部ASCII代码定义的。例如，字符'A'在计算机内部用ASCII代码65表示，在运算时被当作整数65处理。 312

因为整数和字符能自由地相互转换，所以可以很容易地定义一个函数RandomLetter，返回一个随机选择的大写字母。有了第8章的random.h接口导出的RandomInteger函数，它的实现可写为

```
char RandomLetter(void)
{
    return (RandomInteger('A', 'Z'));
}
```

尽管对char类型的值应用任何算术运算都是合法的，但在它的值域内，不是所有运算都是有意义的。例如，在程序中将'A'乘以'B'是合法的。为了得到这个结果，计算机取它的内部代码，65和66，将它们乘起来，得到4290。问题在于这个整数作为字符毫无意义，事实上，它超出了ASCII字符的范围。当对字符进行运算时，仅有很少的算术运算是有用的。这些有意义的运算通常有：

- *给一个字符加上一个整数* 如果c是一个字符，n是一个整数，表达式c + n表示代码序列中c后面的第n个字符。例如，如果n在0～9之间，表达式'0' + n得到的是第n个数字的字符代码。因此，'0' + 5是'5'的字符代码。同样，如果n在1～26之间，那么'A' + n-1表示字母表中第n个字母的字符代码。
- *从一个字符中减去一个整数* 表达式c-n表示代码序列中c前面的第n个字符。例如，表达式'Z'-2的结果是'X'的字符代码。
- *从一个字符中减去另一个字符* 如果c1和c2都是字符，那么表达式c1-c2表示两个字符在代码序列中的距离。例如，如果你再回顾一下表9-1，并计算每个字符的ASCII值，你可以确定'a'-'A'是32。更重要的是，小写字母和与之对应的大写字母之间的距离是固定的，因此'z'-'Z'也是32。
- *比较两个字符* 用某个关系运算比较两个字符的值是常用的运算，经常用来确定字母的次序。例如，如果在ASCII表中，c1在c2前面，那么表达式c1<c2是TRUE。

toupper(ch) 如果ch是小写字母，返回它对应的大写字母；否则，返回ch本身。

虽然ctype.h接口已经提供了tolower和toupper两个函数，但如果你从头实现它们，则能更进一步了解它们的操作。同样，你可以忽略确切的ASCII代码，仅依赖于连续的假设。如果ch包含一个大写字母的字符代码，可以加上一个大写字母和小写字母的内码之间的差，将它转换为小写。然而，这里并没有把这个差作为一个确定的常量，如果把它表示成字符运算'a'-'A'，则程序的可读性更好。因此，你可以用如下代码实现这个tolower函数：

```
char tolower(char ch)
{
    if (ch >= 'A' && ch <= 'Z') {
        return (ch + ('a' - 'A'));
    } else {
        return (ch);
    }
}
```

函数toupper的实现方法类似。

尽管在ctype.h中定义的函数都很容易实现，但使用库函数而不是自己编写的函数是良好的程序设计习惯。这样做有三个主要的理由。

1) 因为库函数是标准的，其他程序员读你编写的程序比较容易。假设这些程序员都有C语言的经验，他们就知道ctype.h中的函数，并确切知道它们的含义。

2) 库函数比你自己写的函数更能保证正确性。因为ANSI C的库被成千上万个客户程序员所使用，实现者为保证函数的正确性作了大量的工作。如果你自己重写库函数，那么带来错误的机会更大。

3) 函数的库实现通常比你自己写的函数效率更高。例如，在ctype.h接口中，库中的机制比本节中给出的实现运行速度更快，通常是快三或四倍。这些高效的实现到底是如何工作的，这已超出本章的介绍范围，重要的是，你可以使用库提高程序的效率。

9.2.8 涉及字符的控制语句

因为char是标量类型，你可以把它用在可以出现整数的所有语句中。例如，如果ch声明为char类型，你可以用下列的for语句执行26次循环，按字母顺序对每个大写字母执行一次： 315

```
for (ch = "A"; ch <= "Z"; ch++)
```

同样，你也可以将字符用在switch语句的控制表达式中。例如，如果参数为元音的话，下面的谓词函数返回TRUE：

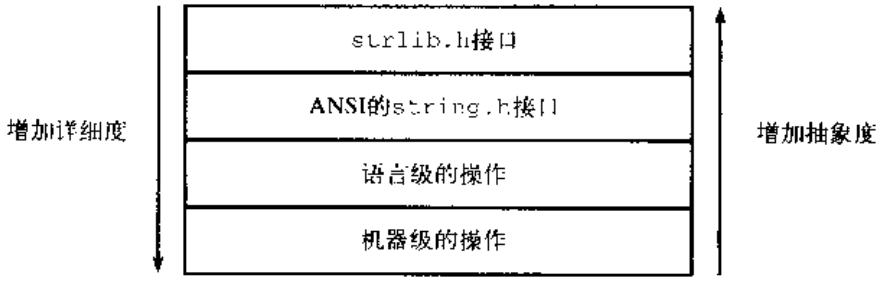
```
bool IsVowel(char ch)
{
    switch (tolower(ch)) {
      case 'a': case 'e': case 'i': case 'o': case 'u':
        return (TRUE);
      default:
        return (FALSE);
    }
}
```

注意，对大写和小写的形式，实现中都用tolower函数识别元音。

9.2.9 字符的输入输出

习惯上，字符的输入输出是用标准I/O库函数getchar和putchar完成的。本书将在第

建立在不同层次结构上的抽象称为**分层抽象**（layered abstraction）。用来表示字符串的分层抽象结构如下图所示： 317



完成输入输出的硬件设备自动在ASCII代码和屏幕或键盘上的符号之间进行转换。如本章前面所讨论，计算机对表示字符的整数代码应用算术运算可以处理每个字符。这些功能组成了可用于字符串的机器级的操作，形成层次结构中的最低一层。

在由硬件提供的基本功能的上面，程序设计语言通常也包括一些对字符串操作的支持。可用于字符串的内嵌操作形成了层次结构中的第二层。在许多语言中，这些功能是非常强大的，可以直接在语言级完成复杂的字符串操作。然而，ANSI C在语言本身几乎没有提供任何对字符串的支持。仅有的支持是提供定义字符串常量功能，这在第2章中的2.5.1节已经进行了介绍。所有其他的字符串操作都是由库提供的。甚至类型名string也不是语言的一部分，而是在genlib.h接口中作为扩展定义的。

在ANSI C中，程序员可用的大多数字符串操作是由string.h接口提供的。这个库提供了一组功能强大的字符串操作，但string.h接口中的概念抽象超出了你目前所学的知识，到学完第11章和第13章的内容后才能理解。然而，某些常用的操作用string.h接口难以完成。例如，当你用string.h中的函数时，不能很容易地从函数返回字符串值，或甚至直接为一个变量赋一个字符串值。因为使用string.h接口是ANSI C中字符串操作的标准方法，最终你必须学会如何使用它。本书将string.h接口放到第14章介绍，确保在你理解这个软件包的概念抽象前，你已经掌握了所有所需的背景知识。

为了让你有机会能用概念上较简单的模型处理字符串，本书引入另一个字符串库，它可以通过strlib.h接口访问。和genlib.h和stdio.h一样，strlib.h接口也是标准ANSI C库的扩展。因此，要把它包含到你的程序中，你必须要用引号：

```
#include "strlib.h"
```
318

strlib.h接口形成了抽象层次结构的最高层，使得字符串操作相对容易。

9.3.2 抽象类型的概念

strlib.h接口的主要优势在于它能使字符串作为*抽象类型*来处理。虽然这个定义将在第17章中细化（在第17章中抽象类型将作为一个中心话题），但目前我们可以把**抽象类型**（abstract type）想象为按照它的行为而不是按照它的表示定义的类型。

抽象类型的行为是由能在这种类型的对象上实施的操作来定义的。特定抽象类型的合法操作被称为它的**基本操作**（primitive operation），被定义为与该类型相关的接口中的函数。这些操作的细节和数据的基本表示都被隐藏在该接口的实现中。不管客户何时要操作某个抽象类型的值，客户必须用该接口提供的函数。

对字符串来讲，所需要完成的基本操作是什么呢？首先，你已经知道如何

下面的程序从用户那里读入一行文本，并报告它的长度：

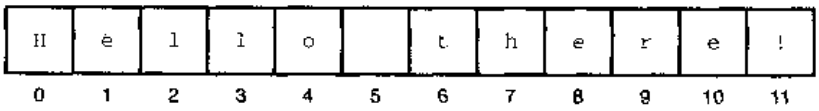
```
main()
{
    string str;

    printf("This program tests the StringLength function.\n");
    printf("Enter a string: ");
    str = GetLine();
    printf("The length is %d.\n", StringLength(str));
}
```

320

9.4.2 从一个字符串中选择字符

在C语言中，字符串中的位置是从0开始编号的。例如，在字符串"Hello there!"中，每个字符的编号如下图所示：



写在字符串的每个字符下面的正数称为它在字符串中的**下标**（index）。

为了使你能按给定的下标在字符串中选择特定的字符，strlib.h接口提供了一个函数IthChar，它有两个参数，一个字符串和一个表示下标的整数，该函数返回一个字符。例如，如果变量str包含字符串"Hello there!"，调用IthChar(str, 0)将返回字符'H'。同样，调用IthChar(str, 5)将返回' '，这是一个空格。记住，C语言中字符的编号是从0开始，而不是从1开始。这条规则很容易被忘记，会假设IthChar(str, 5)返回的是字符串中的第5个字符。IthChar(str, 5)返回的是下标位置为5的字符，是字符串中的第6个字符。

给出了库函数IthChar之后，你可以定义一个新的函数LastChar(str)，它返回str中的最后一个字符，如下所示：

```
char LastChar(string str)
{
    return(IthChar(str, StringLength(str) - 1));
}
```

例如，对字符串"Hello there!"调用LastChar将返回一个感叹号'!'。

9.4.3 连接

字符串的另一个有用的函数是Concat函数，它将两个字符串连接起来，当中不加任何字符。在程序设计中，这个操作被称为**连接**（concatenation）。例如，

```
Concat("Hello", "there")
```

的值是十个字符组成的字符串"Hellothere"。

如果要在两个单词间放一个空格，必须执行另一个连接。Concat函数每次只能取两个参数。为了连接三个或更多的字符串，必须多次调用Concat。每次调用连接两个字符串。例如，如果变量word1包含"Hello"，变量word2包含"there"，为了产生11个字符的字符串"Hello there"，必须嵌套调用Concat，如下列表达式所示：

321

```
Concat(Concat(word1, " "), word2)
```

以反序出现。下面的ReverseString的实现说明了这个策略：

```
string ReverseString(string str)
{
    string result;
    int i;

    result = "";
    for (i = 0; i < StringLength(str); i++) {
        result = Concat(CharToString(IthChar(str, i)), result);
    }
    return (result);
}
```

注意，每个字符在连接到变量result的开始处以前，必须被转换成字符串。

9.4.5 抽取字符串的一部分

连接将几个短的字符串形成长的字符串。你还经常要做相反的工作：将一个字符串分解为较短的字符串。一个字符串如果是某一较长的字符串的一部分，则被称为一个**子串**（substring）。strlib.h库提供了一个函数SubString(s, p1, p2)，它的作用是从s中抽取从位置p1到p2之间的字符，包括p1和p2。因此，函数调用

```
SubString("Hello there!", 1, 3)
```

返回字符串"ell"。如你所知，在C语言中字符编号是从0开始，因此，下标位置为1的字符是字符'e'。

作为SubString的一个应用实例，函数SecondHalf(s)返回一个由s中后半部分字符组成的子串，如果字符串长度是奇数，则包括中间的字符：

```
string SecondHalf(string str)
{
    int len;

    len = StringLength(str);
    return (SubString(str, len / 2, len - 1));
}
```

SubString函数对一些特殊情况处理如下：

323

1) 如果p1是负数，它会被设为0，使它指向字符串的第一个字符。

2) 如果p2大于StringLength(s)-1，它会被设置为StringLength(s)-1，使它指向最后一个字符。

3) 如果p1大于p2，则SubString返回一个空串。

虽然在你对字符串没有更多的经验前可能不太清楚选择这种设计的理由，但当你调用SubString时，这些规则使你不需要检查一些特殊情况，使字符串编程更容易。

9.4.6 比较两个字符串

从第5章开始，你已经能用StringEqual函数比较两个字符串是否完全相同。在许多情况下，你还会发现确定两个字符串按字母顺序的先后位置也是非常有用的。strlib.h接口提供了一个函数StringCompare实现这个功能。StringCompare有两个字符串参数s1和s2，它返回一个整数，该整数的符号指出了两个字符串的关系：

- 如果按字母顺序，s1出现在s2的前面，则StringCompare返回一个负整数。

strlib.h接口提供了两个函数FindChar和FindString来实现这个功能。FingChar的原型是

```
int FindChar(char ch, string text, int start);
```

该函数搜索字符串text，从start指定的下标位置开始，寻找字符ch的第一次出现。如果发现了该字符，FindChar返回这个字符的下标位置。如果在text结束前没有发现这个字符，FindChar返回值-1。下面的例子说明了这个操作（记住，下标号从0开始）：

```
FindChar('l', "Hello there", 0)   (返回2)
FindChar('l', "Hello there", 3)   (返回3)
FindChar('l', "Hello there", 4)   (返回-1)
```

325

和字符串比较一样，搜索字符串的函数对大小写是区别对待的。因此，调用

```
FindChar('h', "Hello there", 0)
```

返回7，因为小写字母h的第一次出现在下标位置7。在下标位置0的大写字母H被忽略了。

你可以用FindChar实现一个产生**首字母缩写**（acronym）的函数，它按次序取一系列单词的首字母形成一个新的单词。例如，单词scuba是由self contained underwater breathing apparatus的首字母形成的首字母缩写词。函数Acronym取一个由若干个独立的单词组成的字符串，返回它们的首字母缩写词。因此，调用函数

```
Acronym("self contained underwater breathing apparatus")
```

将返回"scuba"。

只要单词用单个空格分开，并且没有其他字符出现，Acronym的实现就非常简单，它取最前面的字母，然后进入循环，搜索空格。一旦发现了空格，就将下一个字符连到保存结果的字符串变量的最后。当字符串中找不到空格时，首字母缩写词就完成了。这个策略可以翻译成下列C语言的实现：

```
string Acronym(string str)
{
    string acronym;
    int pos;

    acronym = CharToString(IthChar(str, 0));
    pos = 0;
    while (TRUE) {
        pos = FindChar(' ', str, pos + 1);
        if (pos == -1) break;
        acronym = Concat(acronym,
                         CharToString(IthChar(str, pos + 1)));
    }
    return (acronym);
}
```

函数FindString(str, text, start)与FindChar相似，除了第一个参数是字符串以外。函数搜索字符串text，从位置start开始寻找字符串str。如果找到匹配的字符串，FindString返回匹配开始处的下标位置。例如，

```
FindString("there", "Hello there", 0)
```

将返回值6。如果没有找到匹配的字符串，和FindChar一样，FingString返回-1。

例如，函数ReplaceFirst(str, pattern, replacement)搜索字符串str，将字

果n是负数，前面再加一个负号。例如，调用IntegerToString(123)将返回字符串"123"，调用IntegerToString(-4)将返回字符串"-4"。RealToString(d)函数将浮点数d转换为字符串，该字符串的格式类似于由printf用格式码%G的显示结果，有时会产生一个以科学记数法表示的数字。例如，调用RealToString(3.14)返回“3.14”，而调用RealToString(0.00000000015)将返回"1.5E-10"。

```
/*
 * File: repfirst.c
 * ----------------
 * This file implements and tests the function ReplaceFirst.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

string ReplaceFirst(string str, string pattern, string replacement);

/* Main program */

main()
{
    string str, pattern, replacement;

    printf("This program edits a string by replacing the first\n");
    printf("instance of a pattern substring by a new string.\n");
    printf("Enter the string to be edited:\n");
    str = GetLine();
    printf("Enter the pattern string: ");
    pattern = GetLine();
    printf("Enter the replacement string: ");
    replacement = GetLine();
    str = ReplaceFirst(str, pattern, replacement);
    printf("%s\n", str);
}

/*
 * Function: ReplaceFirst
 * Usage: newstr = ReplaceFirst(str, pattern, replacement);
 * --------------------------------------------------------
 * This function searches through the string str and replaces the
 * first instance of the pattern with the specified replacement.
 * If the pattern string does not appear, str is returned unchanged.
 */

string ReplaceFirst(string str, string pattern, string replacement)
{
    string head, tail;
    int pos;
```

图9-1 repfirst.c

```
        total += StringToInteger(line);
    }
    printf("The total is %d\n", total);
}
```

由于GetInteger不能读一个空行作为数据，因此在第3章的addlist.c中没有将空行作为输入结束标记的一种选择。如果程序用GetLine读入输入值，空行将作为一个空串出现。如果从用户读入的行不是空行，那么该行中的字符将转换成整数，并加到运行结果中。

9.4.10 效率和strlib.h库

本章中所有函数的实现都是以清晰，而不是以效率作为目标。当你学完第14章中各个字符串库的内部细节时，你将发现本章介绍的许多实现的效率都是相当低的，低得无法适应一些重大的应用。然而，它们是简明的、可行的、容易理解的。当用这种形式的函数工作时，你可以从概念上了解字符串工作的过程。当你阅读strlib.h接口的实现并研究它如何工作时，这些知识是非常有帮助的。 330

小结

在本章中，你已经开始理解如何使用*文本数据*。在C语言中，文本数据最常见的形式是*字符串*，它是单个字符的有序集合。单个字符在C语言中用数据类型char表示，它是称为*标量类型*的一类数据的一部分。标量类型也包括*枚举类型*，它也是在本章中介绍的。使用字符是处理字符串的基础，因为字符是形成字符串的“原子”。

在本章中，你学习了如何将字符串作为*抽象类型*来操作。抽象类型的内部表示被隐藏起来。客户可以通过调用在某个接口中定义的函数来使用该类型的值。在字符串的情况中，提供字符串操作的接口是strlib.h，它形成了*分层抽象*的最高层次。通过学习如何将字符串作为抽象数据类型来操作，你可以逐步发现如何以令人兴奋的和复杂的方法使用它们，尽管它的结构是不可见的。

然而，因为字符串是现代程序设计的基础，有必要从不同的角度去了解它。所以在第11章、第13章和第14章还将学到更多的有关字符串以及它们的表示的知识。

本章的重点内容包括：

- 值不是数字的类型在计算机内部通常通过对该类型的值域中的元素编码，然后用这些编号作为原先值的代码来表示。通过对元素进行计数而定义的类型称为*枚举类型*。
- C语言用关键词typedef和enum定义新的枚举类型。
- 枚举类型是*标量类型*的子类，标量类型的行为与整数的行为完全一样。在C语言中，标量类型和整数类型之间不需要显式的转换。
- 按照预先定义的*字符代码*，字符在内部被表示为整数。虽然某些计算机使用不同的字符编码系统，但大多数现代的计算机都是用表9-1所示的ASCII编码系统。
- 用来表示单个字符的数据类型char是一个标量类型。因此，字符值可以用标准的算术运算来处理。
- ctype.h接口包含一些为单个字符分类，及改变单个字符大小写的函数。
- strlib.h接口使字符串能作为*抽象类型*工作。
- strlib.h接口定义了一些处理字符串的函数。这些函数归纳在表9-3中。 331

符，尽管它们的类型是int。执行实现中的这些步骤，验证在表示字符的所有值上完成的操作确实是在上一个问题中列出的这些操作。

12. 调用isdigit(5)的结果是什么？调用isdigit('5')的结果是什么？

13. 调用toupper('5')的结果是什么？

14. 判断题：在switch语句中，使用字符常量作为case表达式是合法的。

15. 什么是分层抽象？

16. 判断题：抽象类型是按其行为，而不是按其表示定义的。

17. 调用下列函数的结果是什么？

a. StringLength("ABCDE")

b. StringLength("")

c. StringLength("\a")

d. IthChar("ABC", 2)

e. Concat("12", ".00")

f. CharToString('2')

g. SubString("ABCDE", 0, 3)

h. SubString("ABCDE", 4, 1)

i. SubString("ABCDE", 3, 9)

j. SubString("ABCDE", 3, 3)

18. 如果你要在一个已有的字符串后添加一个字符，应该用strlib.h中的什么函数？ 333

19. 当比较字符串时，必须牢记哪些重要的注意事项？

20. 调用下列函数的结果是什么？（对调用StringCompare，只需指出结果的符号。）

a. StringEqual("ABCDE", "abcde")

b. StringCompare("ABCDE", "ABCDE")

c. StringCompare("ABCDE", "ABC")

d. StringCompare("ABCDE", "abcde")

e. FindChar('a', "Abracadabra", 0)

f. FindString('ra', "Abracadabra", 3)

g. FindString("is", "This is a test.", 0)

h. FindString("This is a test", "test", 0)

21. 调用下列函数的结果是什么？

a. ConvertToLowerCase("Catch-22")

b. StringToInteger(SubString("Catch-22", 5, 7))

c. RealToString(3.140)

d. Concat(IntegerToString(4/3), "pi")

程序设计练习

1. 定义一个函数LeftFrom(dir)，它有一个类型为directionT的参数，表示指南针上的点，返回一个左转90度后指南针上的点。为你的函数写一个测试程序，显示对指南针上的每个点调用LeftFrom后的结果。如下所示：

写一个主程序测试ScrabbleScore函数。这个测试程序的一次可能的运行示例如下所示：

```
This program tests the ScrabbleScore function.
Enter words, ending with a blank line.
Word: XI↵
The basic score for 'XI' is 9.
Word: HORN↵
The basic score for 'HORN' is 7.
Word: SCRABBLE↵
The basic score for 'SCRABBLE' is 14.
Word: QUIzZICAL↵
The basic score for 'QUIzZICAL' is 28.
Word:
```

6. 实现一个函数Capitalize(str)，它返回一个首字符是大写的（如果是字母），其他所有字母都转换成小写的字符串。其他字符不受影响。例如，Capitalize("BOOLEAN")和Capitalize("boolean")应该都返回字符串"Boolean"。

7. 实现一个谓词函数EqualIgnoringCase(s1, s2)，在不考虑大小写的情况下，如果字符串s1和s2是相同的，则返回TRUE。例如，EqualIgnoringCase("CAT", "cat")应该返回TRUE。

8. 使人们难以读出信息的最简单的编码方法之一是**字母替换加密**（letter-substitution cipher），原始消息中的每个字母都由不同的字母来替换。字母替换加密中最简单的一种是**循环加密**（cyclic cipher）。其中每个字母都用字母表中在该字母之后的、与其相隔固定距离的另一字母替代。单词循环指的是如果操作到字母表开始的字符时，则通过Z再次循环到字母表开始，重新从A开始。

作为一个实例，下面的运行示例显示了字母表中的每个字母向前移4个字母。A变成了E，B变成了F，Z变成了D（因为它循环到字母表的开始位置），等等。

```
This program encodes a message using a cyclic cipher.
Enter the numeric key: 4↵
Enter a message: ABCDEFGHIJKLMNOPQRSTUVWXYZ↵
Encoded message: EFGHIJKLMNOPQRSTUVWXYZABCD
```

336

为了解决这个问题，应该先定义一个函数EncodeString，它的原型为

```
string EncodeString(string str, int key)
```

该函数返回一个新的字符串，它是由将str中的每个字母后移key个字母而形成的，必要时，还要返回到字母表的开始位置。实现EncodeString后，编写一个测试程序，重复下面的运行示例：

```
This program encodes a message using a cyclic cipher.
Enter the numeric key: 13↵
Enter a message: This is a secret message.↵
Encoded message: Guvf vf n frperg zrffntr.
```

注意，编码操作仅用于字母；其他字符在输出时保持不变。此外，字母的大小写也不受影响；小写字母在编码后还是小写字母，大写字母在编码后也还是大写字母。

写一个程序，使得负的key值表示字母向字母表的开始位置移动，而不是向字母表的结

第10章 模块化开发

> It is particularly important, it seems to me, in an era of ever increasing departmentalization and specialization, to make the attempt occasionally to see wholes and to understand what lies behind the exterior manifestations.
>
> — Jessamyn West, *The Quaker Reader*, 1962

学习目标

- 认识把单个程序分为多个单独模块的重要性。
- 理解在一个模块里需要在多个函数调用之间保存状态信息。
- 能够使用全局变量来表示在跨函数调用中所需维护的状态信息。
- 认识到过度使用全局变量的危险性。
- 能够使用static关键字来保证一个模块中的函数和全局变量的私有性。

339

目前为止，本书给出的大部分程序都是一些短小的程序，用来说明C语言的一些特殊方面。关注单个语句格式或者其他一些语言细节对于初学编程的人来说是有意义的。这样读者能够单独考虑每个概念，并且了解它是如何工作的，而不必涉及大程序中的内在的复杂性。但是，真正具有挑战性的则是如何掌握这种复杂性。为了达到这个目的，就必须写一些大的程序，在这样的程序中，将一些单独的概念和工具结合起来。

管理大程序的最重要的技术是第5章所学的逐步精化策略，即当遇到一个需要复杂解决方案的较大的问题时，把问题分解成更为容易解决的几个小部分。

把一个大的问题分解成小片段的策略的方法在程序设计过程的很多阶段都会用到。第5章中的calender.c实例在只有一个源文件的情况下阐述了此项技术。然而，当程序变得更长的时候，要在一个源文件中处理如此众多的函数会变得困难。就像要一下子理解一个50行的函数也是比较困难的一样。在这两种情况下，引入一些额外的结构是很有用的。比如碰到一个50行的函数时，最好的方法是把它分成几个相互调用的小函数来完成任务。当碰到一个包含50个函数的程序时，最好的办法就是把程序再分成几个更小的源文件。每个源文件都包含一组相关的函数。由整个程序的一部分组成的较小的源文件称为**模块**（module）。每个独立的模块比整个程序要简单。而且，如果在设计的时候非常仔细的话，可以把同一模块作为许多不同的应用程序一部分。

program.c

```
main()
{
   ProcA();
   ProcB();
}

void ProcA(void)
{
   . . .
}

void ProcB(void)
{
   . . .
}
```

图10-1 作为单个模块的程序

340

当把一个程序分成模块的时候，选择合适的分解方法来减少模块之间相互依赖的程度是很重要的。包含main函数的模块叫**主模块**（main module），在分解层次中处于最高层。每个其他模块代表一个独立的抽象，其操作在一个接口中定义。

2) 如果单词以元音开头，则加后缀way。

例如，假设单词是scram。因为是以辅音开头，所以把它分为两部分。第一部分由第一个元音之前的所有字符组成，另一部分包括那个元音以及之后的字母，（如下图所示）。



然后交换这两部分的位置，并且在末尾加上ay（如下图所示）。



这样就创建了Pig Latin单词amscray。对于以元音开头的单词，如apple，只要在末尾加上way，从而得到appleway。

10.1.1 应用自顶向下的设计

既然问题是把整行英文翻译成Pig Latin，那么程序应该能够产成如下运行示例：

```
Enter a line: this is pig latin.↵
isthay isway igpay atinlay.
```

342

在开始设计Pig Latin问题的解决方案时，可以先不立即决定是否需要把程序分割成的模块。通常，仅仅在比较了不同的解决方法以后，才可以决定一个程序是否需要一个特定模块分解。通常，最好的方法是先应用自顶向下的设计方法，如5.6节所描述的那样。

采用自顶向下的设计方法，从主程序的层次着手，然后依次写下一系列函数，每个函数解决整个问题中的一个部分。在初始阶段，可以用一系列有待具体实现的高层步骤来定义main函数。虽然可以将这些步骤直接编码为函数调用，但是通常先描述出其功能更为容易。例如，当你在纸张上设计程序时，你可以写出如下的Pig Latin程序的初稿：

```
main()
{
    Read in a line of text from the user.
    Translate the line of text into Pig Latin.
}
```

在这个例子里，函数头和花括号是C语言标准语法的一部分，而语句本身则写成英文语句，用来描述程序要做什么事情。由英文和C语言混合构成的程序称为**伪代码**（pseudocode）。

10.1.2 使用伪代码

虽然伪代码对编译器来说没有意义，但是对于程序员非常有用，因为它可以记录逐步精化的过程。在用一系列英文步骤写出整个程序之后，可以再回到伪代码语句，用实际的C语言代码来实现它们。例如，在这个例子中，可以很容易地把第一句伪代码翻成C语句，因为这和第2章以来我们一直使用的读取一个字符串的习语相吻合。在填写完从用户处读入一行文本的操作之后，程序的伪代码版本如下所示：

```
main()
{
    string line;

    printf("Enter a line:");
```

你学完第11章的时候，你会有用来处理单词列表的工具，但到目前为止，你必须使用现在所具备的工具来完成任务。如果仔细考虑这个问题，会发现不需要立即记录所有的单词。一旦找到一个单词，就可以马上翻译并且正确显示。一旦显示出这个单词，就可以丢开它继续处理下一个单词。根据这个观察结果，可以提出第二种策略：

```
Void TranslateLine(string line)
{
    while(there are any words left on the line){
        Get the next word.
        Translate that word into Pig Latin.
        Display the translated word.
    }
    Display a newline character to complete the output line.
}
```

虽然这个策略的伪代码里还有很多细节问题没有考虑，但是总体思想看起来是有意义的，并且可以避免记录整个单词列表的问题。

10.1.4 考虑空格和标点符号的问题

在TranslateLine的伪代码版本所用的策略中有个小问题。例如，在程序运行时，假设用户输入了以下数据行：

```
Enter a line: this is pig latin.↵
```

如果把输入看作四个单词this、is、pig和latin，如果所有的英文表示的步骤都如预期的那样正常运行，那么程序输出将会是：

```
Enter a line: this is pig latin.↵
isthayiswayigpayatinlay
```

345

这个输出并不是我们真正想要的。因为没有空格和标点符号，所以所有的单词都挤在一起了。这个伪代码版本没有把空格和标点符号考虑进去。而原来对问题的英文陈述也没有考虑这个问题。这个问题没有被完整地定义过。

程序设计中的一个现实问题是问题的英语描述通常是不完整的。作为一个程序员，常常会被一些所忽略的或者认为太明显而没有提及的细节所难倒。在某些情况下，被省略的部分是非常重要的，需要跟给你布置任务的人进一步讨论。然而，在很多时候，你必须自己选择一种合理的策略。

但是，判断什么是合理的策略也有一些技巧。在这个例子中，需要决定的是在输出的每个单词之间打印空格，而忽略其他标点符号。这个策略很简单，而且在本例中很合理。但另一方面它又可能不是最好的。标点符号可以提高输出的可读性。因为标点符号和空格都能表达出某种意义，所以最好让它们在输出中的位置和在输入中的位置相同。因此，需要的输出为：

```
Enter a line: this is pig latin.↵
isthay isway igpay atinlay.
```

```
        Get the next token.
        if (the token is a regular English word) {
            Replace the token by its Pig Latin translation.
        }
        Display the token.
    }
    Display a newline character to complete the output line.
}
```

这个策略仍然是相当简单的。而且，获取一个记号并测试这些记号是否留在该行中的操作很可能在不同的应用程序中都是有用的。

把一行分成一些独立的记号的想法在计算机科学中非常常见。例如，当C编译器将一段程序翻译成机器代码的时候，此过程的第一步就是将输入文件分割成C语言所用的记号，即变量名、数字、运算符等等。将输入分解成记号的过程称为**词法分析**（lexical analysis），或者更通俗地说，把它称为**记号扫描**（token scanning）。

10.1.6 设计记号扫描器

要使用新的策略来完成TranslateLine的实现，必须首先设计能够分割输入行的记号扫描器。而且，要记住它很可能是个通用的工具，除了将字符串转换成Pig Latin外，记号扫描器还能用于许多其他问题。因此，将记号扫描器设计成一个独立的模块就十分有意义。起初，设计过程需要考虑如何组织扫描器模块，特别是它该包含什么函数。

在前面一节给出的最新的TranslateLine伪代码版本中，扫描器模块有两个不同的作用。第一，扫描器必须提供一个函数返回当前行中的下一个记号。第二，当扫描完最后一个记号时要让客户知道。因为你正在设计这个接口，所以你要选择这些函数的名字。负责返回下个记号的函数可以命名为GetNextToken。要报告所有记号都已经被扫描，一种选择是定义一个叫AtEndOfLine的谓词函数，它在读到最后一个记号时返回TRUE。

但现在你面临一个有趣的问题：这些函数的参数是什么呢？初一看，调用程序必须将输入行传递给GetNextToken，因为记号是来自于那一行的。另一方面，调用GetNextToken(line)的想法在概念上有矛盾之处。如果GetNextToken像通常的函数那样运作，那么GetNextToken(line)每次都返回同样的结果，因为line的值没有改变。

为了解释这个问题，我们假设把字符串"Hello there"分成三个记号：单词"Hello"，跟在它后面的空格，以及单词"there"。另外，假设已经用赋值语句

```
line = "Hello there";
```

把字符串"Hello there"存储在变量line中。如果GetNextToken将line作为参数，很容易就能想到调用GetNextToken(line)将得到第一个单词。问题是如何得到下一个记号。如果你再次调用GetNextToken(line)，变量line仍然包含着整个字符串"Hello there"。既然这个函数调用跟第一次出现的形式完全相同，那么它将返回同一个值。

为了改正这个问题，必须将扫描器模块设计为能够追踪划分行的进度。在GetNextToken从行返回一个记号后，必须记住已经扫描过的那些记号，以便在下次调用时返回一个不同的结果。一个模块中的多次函数调用之间的信息称为**内部状态**（internal state）。

当模块维护内部状态时，模块的接口通常导出一个用来初始化此状态信息的函数。例如，在扫描器模块中，提供一个函数InitScanner(line)是有用的，它使得客户能够告诉扫描器在字符串line的开头开始返回记号。为了得到这些记号，客户只要调用GetNextToken

```
bool IsLegalWord(string token)
{
    int i;

    for (i = 0; i < StringLength(token); i++) {
        if (!isalpha(IthChar(token, i))) return (FALSE);
    }
    return (TRUE);
}
```

翻译一个单词的过程稍微有些难。在伪代码中，TranslateWord的结构对应了Pig Latin的规则。

```
string TranslateWord(string word)
{
    Find the position of the first vowel.
    if (the vowel appears at the beginning of the word) {
        Return the word concatenated with "way".
    } else {
        Extract the initial substring up to the vowel and call it the "head."
        Extract the substring from that position onward and call it the "tail."
        Return the tail, concatenated with the head, concatenated with "ay".
    }
}
```

350

在伪代码中，第一条英文语句是最难实现的，而且是唯一对于定义一个新函数有意义的语句。函数FindFirstVowel(word)返回word的第一个元音的下标位置。这样，在TranslateWord实现中的第一条语句可以写成如下形式，其中vp是个用来记录元音位置的整型变量：

```
vp-FindFirstVowel(word);
```

如果你在设计中非常仔细，就会注意到函数FindFirstVowel(word)还没有完整的定义。非形式化的描述没有包含所有可能的情况。如果word中没有元音怎么办？在这种情况下，FindFirstVowel仍然要返回一些东西。库函数FindChar和FindString使用－1作为一个特殊的标志，用来指出函数查询的值在字符串中不存在。这里也可以采用相同的策略。

当FindFirstVowel返回－1时，TranslateWord必须有合理的响应，因为转换一个不包含元音的单词的结果还没有明确定义过，因此需要确定TranslateWord到底该做什么。最简单的办法是让TranslateWord返回原来的单词。结合这个设计决策意味着TranslateWord的代码将以如下行开始：

```
vp = FindFirstVowel(word);
if (vp == -1) {
    return (word);
} else . . .
```

TranslateWord接下来的步骤只是从strlib.h接口中调用合适的函数，这样使得编码过程很容易完成，如下所示：

```
string TranslateWord(string word)
{
    int vp;
    string head, tail;

    vp = FindFirstVowel(word);
    if (vp == -1) {
```

```
 *
 * When the last token has been read, the predicate function
 * AtEndOfLine returns TRUE, so that the loop structure
 *
 *        while (!AtEndOfLine()) {
 *            token = GetNextToken();
 *            . . . process the token . . .
 *        }
 *
 * serves as an idiom for processing each token on the line.
 *
 * Further details for each function are given in the
 * individual descriptions below.
 */

#ifndef _scanner_h
#define _scanner_h

#include "genlib.h"
/*
 * Function: InitScanner
 * Usage: InitScanner(line);
 * -------------------------
 * This function initializes the scanner and sets it up so that
 * it reads tokens from line.  After InitScanner has been called,
 * the first call to GetNextToken will return the first token
 * on the line, the next call will return the second token,
 * and so on.
 */

void InitScanner(string line);

/*
 * Function: GetNextToken
 * Usage: word = GetNextToken();
 * -----------------------------
 * This function returns the next token on the line.
 */

string GetNextToken(void);

/*
 * Function: AtEndOfLine
 * Usage: if (AtEndOfLine()) . . .
 * -------------------------------
 * This function returns TRUE when the scanner has reached
 * the end of the line.
 */

bool AtEndOfLine(void);

#endif
```

图 10-3 (续)

出现后的其余部分。

和局部变量不同，全局变量以某一种方式保持在内存中，在这种方式下它的值不受函数调用的影响。仍以索引卡片来比喻，全局变量被保存在一个始终有效的独立的卡片上，好像被粘在桌面上，永远都不会被包含局部变量的卡片覆盖一样。模块中每个函数都可以看到全局变量卡片的那些变量。而且，当函数返回时，这些变量值不会丢失。全局变量保持相同的值直到给它赋予新值为止。 355

10.2.2 使用全局变量的危险性

许多程序员觉得全局变量的想法非常有吸引力。毕竟，全局变量可以被程序的所有部分看到。不需要将它们作为参数来传给某个函数。但是，程序员很快会发现使用全局变量会使得代码很难读懂。

奇怪的是，新手觉得优越的特性往往是有经验的程序员非常认真对待的问题。新手喜欢全局变量可以被源文件中任何函数操作的特点。而对有经验的程序员来说，这是一个危险的信号。例如，你在查找一个由于变量被错误赋值而导致的程序错误。如果是个全局变量，由于模块中的每个函数都能操作那个变量，那么问题可能存在于源文件的任何位置。而局部变量的错误位置则更容易确定。如果局部变量值错误，程序员只需要查找应用该变量的函数即可。

为了避免这样的问题，在结构良好的程序中，一般很少使用全局变量。它们的主要优点在于使得像扫描器模块这样的模块可以维护内部状态。既然全局变量在函数调用之间可以维护它们的值，那么对于这样的用途非常理想。于是，在扫描器模块中，你可以用全局变量来追踪传给InitScanner的行和此行的当前位置。

常见错误 *在编写程序时，要避免使用全局变量，因为这样会使程序难以理解、难以调试。只有当模块必须维护函数调用之间的内部状态时，才必须使用全局变量。*

10.2.3 保持变量的模块私有化

全局变量在一个模块的任何地方都可见只是使用它们的问题之一。除非明确地声明，否则C编译器假定其他模块也可以看到全局变量。这样，当声明了一个全局变量后，可能改变其值的函数不只是局限在一个模块中。该变量可能被整个程序的任何模块引用。

在一个结构良好的程序中，独立的模块之间通过在模块间传递参数的函数调用来交换数据。在大多情况中，尤其是刚开始养成设计程序习惯的时候，最好确保每个全局变量不会被一个以上的模块引用。为了避免两个模块引用同一个全局变量的可能性，应该在声明前用关键字static来彻底避免这种危险。如：

```
static int cpos;
```

这个声明定义cpos为一个全局整型变量，在所定义的模块里的任何地方都可见。然而，cpos对于别的模块是无效的，因此它是当前模块私有的。在本书中，所有的全局变量都用static来声明。

在C语言中，单词*static*用来指示变量如何存储，这会在第11章中讨论。而在大多数实际应用中，最好认为*static*就是*prviate*的同义词，这样能更贴切地描述它的用途。用关键词static 356
声明变量可以使它们对于应用它们的函数来说是私有的。

10.2.4 初始化全局变量

在scanner.h接口中，通过调用InitScanner函数来初始化全局变量。使用一个初始

指出客户想要的扫描器的常规行为，即返回大小写混合的记号，就和输入行中出现的这些字符一样。

然而，这种方法有两个严重缺点。首先，程序阅读者很难知道在调用InitScanner时TRUE和FALSE参数的含义。为了理解这些参数的用途，任何客户都不得不阅读接口注释。第二，新的设计改变了已有的接口而破坏了稳定性。如果扫描接口已经有客户，那么这些客户不得不修改程序。

避免以上两种问题的更好的办法是扩展扫描器接口而非改变它。所有老的函数都像以前那样工作。为了提供返回大写记号的选项，可以增加一个新的函数ReturnUppercaseTokens，它带有一个布尔值，用它来设置uppercaseFlag。因此，调用

```
ReturnUppercaseTokens(TURE);
```

客户可以选择返回大写记号的新的行为模式。

要注意的是，如果ReturnUppercaseTokens从来未被调用，那么模块必须按照以前那样继续工作。毕竟，使用老的scanner.h接口的现有程序中没有一个程序会调用ReturnUppercaseTokens。甚至写这些客户程序时，这个函数还不存在。因此，即便没有明确的调用来设置它的值，变量uppercaseFlag必须用FALSE作为其初值。为了保证它有恰当的值，需要使用静态初始化： 358

```
static bool uppercaseFlag = FALSE;
```

本章的练习8给你一个机会来完成ReturnUppercaseTokens扩展的实现。

除非客户专门采取行动去改变否则一直在程序里使用的值称为**默认值**（default value）。在一个典型的模块中，指定客户可以设置的选项的全局变量通常被静态初始化为其默认值。客户通过调用接口提供的函数来改变这些值。

10.2.5 私有函数

关键字static除了用于全局变量外，还可以用来指示某函数是某个特定模块的私有函数。定义接口时候，接口导出的函数不是私有的。接口的要点就是让这些函数可以在其他模块中调用。在很多情况下，接口还包括一些只能在当前模块中调用的函数。要指出某一函数是否被限制在一个特定的模块中，可以把关键字static放在函数原型和其实现的前面。这样使得客户无法调用这些函数，从而使接口与用户间的抽象边界更加坚固稳定。

声明函数为static在由几个程序员参与开发的大型程序环境中也有好处。如果函数或者全局变量没有声明为static，组成整个程序的模块集合中的其他模块就不能使用这些名字。在不同函数中，避免名字相互干扰就需要独立模块的开发者之间必须相互沟通，确保不使用相同的名字，或者使用static关键字来保证他们使用的名字对于自己的模块的私有化。

如下的规则对于模块化开发来说是极好的指导：

> **静态声明规则** 除了main函数和接口明确导出的函数之外，所有的函数必须被声明为static。

从现在开始，本书所有的程序都遵循静态声明规则。

10.3 实现扫描器抽象

一旦理解模块的内部状态可以用全局变量保存，那么实现扫描器抽象的实际流程就变得

```
/*
 * File: scanner.c
 * ---------------
 * This file implements the scanner.h interface.
 */

#include <stdio.h>
#include <ctype.h>

#include "genlib.h"
#include "strlib.h"
#include "scanner.h"

/*
 * Private variables
 * -----------------
 * buffer  -- Private copy of the string passed to InitScanner
 * buflen  -- Length of the buffer, saved for efficiency
 * cpos    -- Current character position in the buffer
 */

static string buffer;
static int buflen;
static int cpos;

/*
 * Function: InitScanner
 * ---------------------
 * All this function has to do is initialize the private
 * variables used in the package.
 */

void InitScanner(string line)
{
    buffer = line;
    buflen = StringLength(buffer);
    cpos = 0;
}

/*
 * Function: GetNextToken
 * ----------------------
 * The implementation of GetNextToken follows its behavioral
 * description as given in the interface: if the next character
 * is alphanumeric (i.e., a letter or digit), the function
 * searches to find an unbroken string of such characters and
 * returns the entire string.  If the current character is not
 * a letter or digit, a one-character string containing that
 * character is returned.
 */

string GetNextToken(void)
{
    char ch;
    int start;

    if (cpos >= buflen) Error("No more tokens");
```

361

图10-4 scanner.c

```
/* Main program */

main()
{
    string line;

    printf("Enter a line: ");
    line = GetLine();
    TranslateLine(line);
}

/*
 * Function: TranslateLine
 * Usage: TranslateLine(line);
 * ---------------------------
 * This function takes a line of text and translates
 * the words in the line to Pig Latin, displaying the
 * translation as it goes.
 */

static void TranslateLine(string line)
{
    string token;

    InitScanner(line);
    while (!AtEndOfLine()) {
        token = GetNextToken();
        if (IsLegalWord(token)) token = TranslateWord(token);
        printf("%s", token);
    }
    printf("\n");
}

/*
 * Function: TranslateWord
 * Usage: word = TranslateWord(word)
 * ---------------------------------
 * This function translates a word from English to Pig Latin
 * and returns the translated word.
 */

static string TranslateWord(string word)
{
    int vp;
    string head, tail;

    vp = FindFirstVowel(word);
    if (vp == -1) {
        return (word);
    } else if (vp == 0) {
        return (Concat(word, "way"));
    } else {
        head = SubString(word, 0, vp - 1);
        tail = SubString(word, vp, StringLength(word) - 1);
        return (Concat(tail, Concat(head, "ay")));
```

363

图 10-5 (续)

小结

在本章中，介绍了一些新的技术，利用这些技术可以容易地将一个应用分成多个独立的模块来构成整个程序。特别是如何在模块中表示内部状态，这使得模块中的函数可以在函数调用之间保持某些效用。

本章的重点内容包括：

- 大型程序可以通过分割成独立的模块来加以简化。每个模块写成一个单独的源文件。包含main函数的模块是主模块。
- 当试图通过自顶向下的设计来解决问题的时候，对程序进行有效的模块分解是十分必要的。
- 设计函数时，最好把各个步骤用英文和C语言写出来，这种代码被称为伪代码。
- 模块常常需要维护独立函数调用之间保持的内部状态。
- 内部状态必须用全局变量来保存。一个模块中的所有函数都可以访问全局变量。和局部变量不同，全局变量在函数返回时不会消失。
- 使用全局变量会使程序变得难以阅读和调试。为了避免这些问题，应该仅在需要维护模块内部状态时候使用全局变量。
- 所有用来记录内部状态的变量都应该声明为在其所出现的模块中是私有的。可以通过在变量声明的开始处使用static关键字来实现。
- 除非函数是由接口导出的。否则应该使用static关键字来声明该函数是当前模块中私有的。

复习题

1. 判断题：每个模块对应于一个独立的源文件。
2. 怎么判断几个模块中哪个是主模块？
3. 判断题：在开始实施自顶向下的设计之前，总是该选择对程序进行模块分解。
4. 术语伪代码是什么意思？
5. 当发现问题没有被完整定义的时候该怎么做？
6. 什么是记号？ 366
7. 局部变量和全局变量之间的主要区别是什么？
8. 阅读程序的时候，如何判断变量是局部的还是全局的？
9. 局部变量和全局变量的作用域分别是什么？
10. 为什么过度使用全局变量是有危险的？
11. 关键字static的功能是什么？
12. 静态初始化和动态初始化的区别是什么？什么时候使用静态初始化最有用？

程序设计练习

1. 在几个世纪以前，缩写的形成和现在不同。一种古老的缩写风格不是由单词的首字符组成，而是由前面的辅音（或者如果单词以元音开头则取第一个字母），再跟上单词最后的字母组成。这种风格在美国很多州名的缩写（不是两个字母的邮政缩写）中很常见，例如：

Connecticut	Ct.	Maine	Me.
Florida	Fla.	Pennsylvania	Pa.

6. 写一个函数LongestWord(line)，返回line中最长的单词。用一个可以重复以下示例的主程序来测试这个函数。

```
Enter a line:
All mimsy were the borogoves.↵
The longest word is "borogoves".
```

7. 写一个函数ReverseSentence(line)，把line分成单词并返回由相同单词倒序组成的字符串。函数应该丢弃所有line中的标点符号，并在结果中用一个空格分隔每个单词。例如，主程序读入一行，然后调用ReverseSentence显示结果，可能产生如下输出：

```
Enter a line: double, double toil and trouble:↵
trouble and toil double double
```

8. 扩展scanner.h接口，使它包括在10.2.4节中提到的ReturnUppercaseTokens函数。写一个测试程序通过读入一行数据并且显示每个记号来测试你的扫描器，每个记号占一行。如以下运行示例所示：

```
Enter a line: Hello there.↵
The tokens are:
   "HELLO"
   " "
   "THERE"
```

确保piglatin.c程序不需任何内部改动，就可以使用这个扩展的扫描器模块。

9. 在C程序中，空格和Tab只用来分隔独立的记号而没有其他意义。如果要为C编译器写一个记号扫描器，丢弃由全部空白空格组成的记号是有用的。设计一个scanner.h接口的扩展，允许客户指定空格、Tab和任何其他在ctype.h中的isspace函数返回TRUE的字符被忽略掉。确保你的设计不会影响任何scanner.h接口的任何现有客户。

10. 写一个程序calc.c，实现一个简单的算术计算器。计算器的输入包含由整型常量构成的行，常量由五个C语言中使用的算术运算符（+、-、*、/和%）分隔。对每行输入，程序应该显示每个运算符的运算结果。为了读入单独的值和运算符，你该使用第9题中扩展的扫描器模块来忽略空格。当用户输入一个空行时，程序应该退出。 369

为了降低程序的复杂性，计算器应该忽略C语言的优先级规则，而采用从左到右的运算次序。这样，表达式2 + 3 * 4该得到20而不是C语言中的14。

下面是calc.c的程序的运行示例：

```
This program implements a simple calculator.
When the > prompt appears, enter an expression
consisting of integer constants and the operators
+, -, *, /, and %. To stop, enter a blank line.
> 2 + 2↵
4
```

```
The primes between 0 and 200 are: 2, 3, 5, 7, 11, 13,
17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
193, 197, 199.
```

371

```
/*
 * fill.h
 * ------
 * This file is the interface to an abstraction that enables
 * you to display filled output.  To use the package, you first
 * call SetFillMargin to set an output margin and then make
 * calls to PrintFilledString to display the output.  These
 * functions are described in more detail below.
 */

#ifndef _fill_h
#define _fill_h

#include "genlib.h"

/*
 * Function: SetFillMargin
 * Usage: SetFillMargin(margin);
 * -----------------------------
 * This function sets the fill margin to the value given by the
 * margin parameter.  When strings are displayed using the
 * function PrintFilledString, the output will be broken up
 * into separate lines so that no lines extend past the margin.
 * If no margin is set explicitly, the package uses 65 for the
 * fill margin as a default.
 */

void SetFillMargin(int margin);

/*
 * Function: PrintFilledString
 * Usage: PrintFilledString(str);
 * ------------------------------
 * The string str is displayed on the screen, starting from
 * where the last string left off.  Spaces and newline characters
 * in the string are treated specially.  If a space character
 * appears in an argument, the function may choose to divide
 * the string at that point by moving to the next line.
 * Such division occurs when the next word would not fit
 * within the margin established by SetFillMargin.  A newline
 * character in the string forces the cursor to advance to the
 * beginning of the next line.
 */

void PrintFilledString(string str);

#endif
```

图10-6 fill.h

372

第三部分

复合数据类型

概述

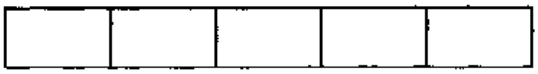
虽然你已经知道程序操作数据，但本书前面部分重点讨论执行操作的控制结构，而不是基本的数据。在第11章～第17章，你将会看到，和控制结构一样，数据结构也是程序设计的中心。通过学习将数据组装成较大的结构，能极大地提高程序设计的能力，并能写出更有用和更激动人心的程序。

1) 数组是有序的。你必须能把数组中的每个分量按顺序排列：这是第一个，这是第二个，等等。

2) 数组是同质的。数组中的每个数值必须有同样的类型。因此你可以定义一个整型数组或是一个浮点型数组，但是一个数组中不能有两种数据类型。

376

直观上，我们可以把数组看成一系列的方框，数组中的每个数值占一个方框。数组中的每一个数值称为一个**元素**（element）。例如，右图代表一个有五个元素的数组。



在C语言中，每个数组有两个基本特性：

- **元素类型**（element type） 存储在数组元素中的数值的类型。
- **数组大小**（array size） 数组所包含的元素个数。

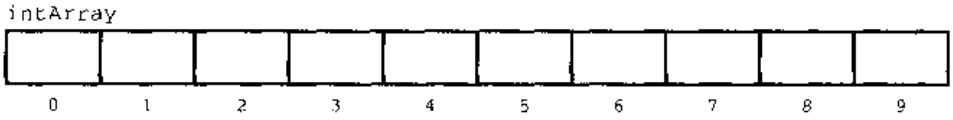
当你在程序中建立一个新的数组时，你必须指明数组的元素类型和数组大小。

11.1.1 数组声明

和C语言中其他的变量一样，数组在使用前必须声明。下面的语法框中给出数组声明的一般形式。例如：

```
int intArray[10];
```

声明了一个名为intArray，且拥有十个元素的数组，每个元素的类型都是int型的。你可以用下面图示的方法描述这种定义，画出10个方框，且命名为intArray：



数组中的每一个元素都由对应的一个叫做**下标**（index）的数值确定。在C语言中，数组的下标从0开始到比数组大小小1的数为止。因此，一个数组有10个元素，那么下标就是0、1、2、3、4、5、6、7、8和9，如上图所示。

语法：数组声明

elementtype arrayname[*size*];

其中：

*elementtype*是数组中每个元素的类型。

*arrayname*是声明为一个数组的变量名。

*size*作为数组一部分的元素数。

在大多数情况下，应该用一个符号化的常量而不是一个具体的数值来指定数组的大小。用一个符号化的常量使得程序员将来可以容易地修改数组的大小。例如，定义数组不能用下列方法：

```
int intArray[10];
```

377

而应该将大小定义成常量：

```
#define NElements 10
```

然后把声明写为

```
int intArray[NElements];
```

11.1.3 一个简单的数组实例

图11-1中给出一个简单的数组操作的实例gymjudge.c。该程序要求用户输入每个裁判员给出的分数，然后显示平均分数。 379

```
/*
 * File: gymjudge.c
 * ----------------
 * This program averages a set of five gymnastic scores.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * ---------
 * NJudges -- Number of judges
 */

#define NJudges 5

/* Main program */

main()
{
    double gymnasticScores[NJudges];
    double total, average;
    int i;

    printf("Please enter a score for each judge.\n");
    for (i = 0; i < NJudges; i++) {
        printf("Score for judge #%d:  ", i);
        gymnasticScores[i] = GetReal();
    }
    total = 0;
    for (i = 0; i < NJudges; i++) {
        total += gymnasticScores[i];
    }
    average = total / NJudges;
    printf("The average score is %.2f\n", average);
}
```

图11-1 gymjudge.c

使用前一节例子中的数据运行程序gymjudge.c，你会得到如下结果：

```
Please enter a score for each judge.
Score for judge #0:  9.2↵
Score for judge #1:  9.9↵
Score for judge #2:  9.7↵
Score for judge #3:  9.0↵
Score for judge #4:  9.5↵
The average score is 9.46
```

380

11.1.4 改变下标值的范围

在C语言中，每个数组的第一个元素的下标值是0。然而，在许多情况下使用这种设计方式

为了弄清数据在计算机中是如何存储的，先要仔细地了解一下在一个典型的机器中内存系统是如何工作的。计算机的内存系统在1.3节中有所涉及，但我们还是不知道内存是如何工作的。

11.2.1 比特、字节和字

从本质上讲，计算机中所有的数值都是按照被称为比特的基本单元的形式存储的。1个**比特**（bit）精确记录一个数值，它可能取两种状态中的一个。如果把机器中的电路当成一个小的开关，那么可以把这两种状态标识为断开和闭合。单词bit实际上是binary digit的缩写，因此把这两种状态用0和1表示就很顺理成章了，0和1这两个数字是计算机运算所基于的二进制系统所用的两个数字。

因为1个比特包含的信息量极为有限，所以比特本身并没有为存储数据提供一个很方便的机制。为了方便地存储整型和字符型等常见类型的信息，将若干个独立的比特组成一个较大的单元，作为完整的存储单元。这种最小的组合单位被称为**字节**（byte），它足以容纳一个需要8个比特的字符[⊖]。在大多数计算机中，字节也可以组成更大一些的称为**字**（word）的结构，一个字的大小通常要求能够容纳一个整型数据。有些计算机用两个字节组成字，有些计算机用四个字节组成一个字，也有很少的计算机的规定十分特殊。

某台计算机可用的内存容量变化很大。内存大小通常用千字节（kilobyte，KB）和兆字节（megabyte，MB）来衡量。在大多数领域中，前缀kilo和mega分别代表1000和1000 000。然 382
而，对计算机而言，十进制系统与机器的内部结构是不匹配的，因此只是用这些前缀表示接近于原来的数值的2的某个整数幂。因此，在程序设计中，这些前缀真正表示的意思应该是：

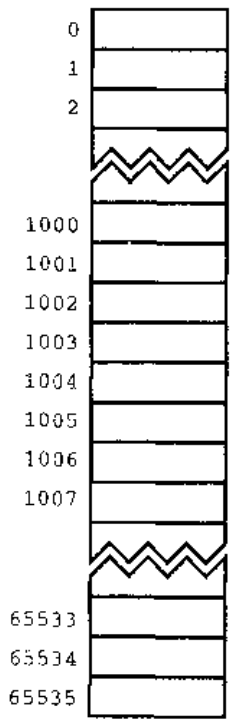
$$\text{kilo(K)} = 2^{10} = 1024$$

$$\text{mega(M)} = 2^{20} = 1\,048\,576$$

一个64KB的计算机（用现在的标准来衡量属于小型的机器）的内存容量是64×1024，即65 536字节。同样，一个8MB的工作站有8×1048576，即8 388 608字节的容量。

11.2.2 内存地址

在存储系统中，每个字节都会有一个数字的**地址**（address）。一般来说，第一个字节在计算机中的地址为0，第二个字节的地址为1，依此类推，直到机器中所含有的字节数为止。例如，你可以用右图表示64KB的计算机的内存。内存中的每一个字节可以容纳一个字符的信息。例如，如果你声明一个字符变量ch，编译器会在当前的函数帧中为这个变量保留一个字节的空间。假设这个字节的地址为1000。如果程序执行语句：




```
ch = 'A';
```

字符'A'的内部表示就会被存储在地址为1000的单元中。由于'A'的ASCII码是65，因此内存的分配结果如下页图a所示。 383

⊖ 虽然几乎在所有的现代计算机中，一个字节由8个比特组成，但ANSI C并没有要求一个字节一定要那么长。仅有的规则是char类型的对象被定义为占据一个字节。因此，一个字节的长度必须足以容纳一个字符代码。

空间。局部变量只有当函数调用时才会分配内存空间。变量本身在分配给这个函数的存储空间中分配，函数的存储空间叫做函数的**帧**（frame）。只要函数在运行，局部变量在帧中的地址始终是不变的。当函数返回时，帧以及它的所有变量都被丢弃，以便让别的函数使用。

如果你声明了一个简单变量，编译器就会为这个变量预留其需要的内存空间，即是sizeof的返回值。如果你声明了一个数组变量，编译器就会为数组元素预留一段连续的内存空间保存组成数组的所有元素的值。根据元素的类型，数组的每个元素可以需要1个字节的内存空间，也可以需要几个字节的内存空间。例如，变量声明：

```
char charArray[20];
```

将会预留20个字节的内存空间，因为每一个字符仅占一个字节。

有些数据类型的每个元素要占有一个以上的字节。例如，假设上述NJudges被设为5，数组中每个元素都是double型的，需要八个字节的内存空间，用下列语句声明的数组scores

```
double scores[NJudges];
```

需要

$$8\text{字节/元素} \times 5\text{个元素} = 40\text{字节}$$

为了存放整个数组，编译器要在当前帧中分配40个字节。如果帧在内存中的开始地址是1000，数组元素的存放情况将如图11-2所示。0号元素将存放在1000～1007，1号元素将存放在1008～1015，依此类推。

当C语言编译器遇到一个选择表达式：

```
scores[i]
```

它通过把下标值和每个元素的大小相乘再加上数组的首地址，来计算出数组元素的地址。在图11-2中，元素的大小是8，数组的首地址是1000。因此，如果i的大小为2，那么scores[i]地址就为

$$2 \times 8 + 1000$$

这个地址是1016，正好与图中scores[2]的地址相匹配。在这种情况下，数组元素的首地址（就是1000）也叫做数组的**基地址**（base address），而找到正确元素所需的调整量（就是2×8，即16）叫做**偏移量**（offset）。

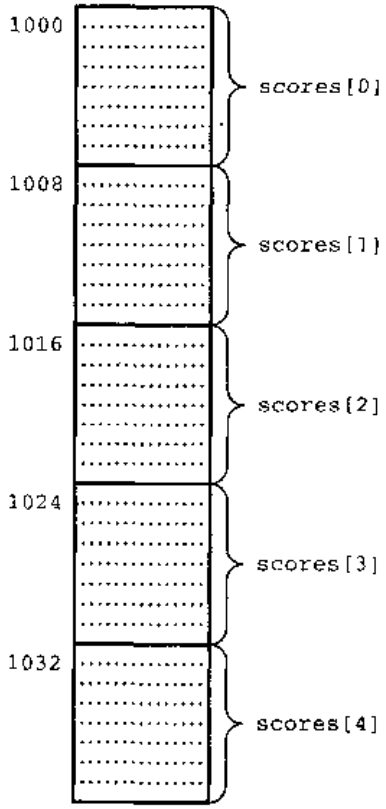


图11-2 scores数组的内存布局

11.2.5 引用超出数组范围的元素

在图11-2中，数组scores有五个合法的下标值：0、1、2、3和4。如果你选择一个下标值超出这个范围的元素，那将会有什么情况发生呢？例如，你要求出表达式

```
scores[i]
```

的值，但这是i的值恰好是5。因为在数组中没有scores[5]，这个结果自然就没有意义。在这种情况下，C语言会如何处理呢？

如果你很幸运使用的是可以防止粗心的程序员犯此类错

函数都小到足以作为一个单元来理解。函数之间通过传递参数来交换信息。如果一个大的程 388
序里面含有数组，那么在分解这个大的程序时就需要函数把整个数组作为参数进行传递。在C语言中，把数组作为参数进行传递和数组在内存中的表示方法有很大关系，有时这不太容易让人理解。在熟悉内部表示法之后，就可以学会数组参数是如何工作的，以及如何有效地使用它们。

将数组作为函数参数的问题可以通过一个简单的例子进行说明。假如让你编写一个名为reverse.c的程序，来实现下面的功能：

1) 读入一串整型数据，直到输入0作为输入结束标志为止。

2) 把这些整型数据逆序排列。

3) 输出经过重新排列后的数据。

下面的运行示例说明了这个程序的操作过程：

```
? 1↵
? 2↵
? 3↵
? 4↵
? 5↵
? 0↵
5
4
3
2
1
```

为了说明数组作为参数的传递过程，重要的是把程序分解成三个函数，分别对应于程序的三个阶段：读入输入数据并将其保存到数组中，对数组元素逆序排列，然后输出结果。用这种方法分解程序的话，主函数就会有如下的结构：

```
main()
{
    int list[NElements];

    GetIntegerArray(list);
    ReverseIntegerArray(list);
    PrintIntegerArray(list);
}
```

（这个程序只是逆序表程序的一个大概的轮廓。最终的版本有不同的参数结构）

然而，这种通用的设计方法会产生两个问题：

1) 在编写的时候，数组元素的个数是用常量Nelements确定的，但在问题说明中，要求元素的个数在用户输入结束标志0之后才能确定。 389

2) 函数GetIntegerArray和ReverseIntegerArray只有在它们改变了参数数组中的数值时才有用。然而，到现在为止，在本书中看到的函数还都不能改变它们的参数值。如果数组和简单变量一样的话，那么整个程序的设计就必须改变了。

上述问题将会在下面几节中具体介绍。

11.3.1 元素个数的通用化

从问题说明中可以看出，程序reverse.c并不能事先确定数组元素的个数，而是让用户输入任意个值，用一个预先设定好的输入结束标志值表示数组结束。这种设计方法对用户来

参数，依此类推。在这个例子中，数组名在主程序中是list，而在子函数中是array。选用不同的名字使你容易区分在主程序中和其他函数中的数组。

因为ReverseIntegerArray函数也需要访问数组数据和有效长度，所以它和PrintIntegerArray有同样的参数。它的函数原型为如下形式：

```
void ReverseIntegerArray(int array[ ], int n);
```

函数GetIntegerArray却有不同的结构。主程序是不可能传递数组的有效长度给GetIntegerArray函数的，在函数GetIntegerArray被调用的时候，主程序是不知道数组的有效长度的。当用户输入数据时，函数GetIntegerArray通过计数确定数组的有效长度，直到用户输入了结束标志值为止。此时，GetIntegerArray必须告诉主程序输入了多少个数值，因此它是返回一个整型数值的函数。

尽管GetIntegerArray函数不需要将数组的有效长度作为参数，但它需要知道数组的分配长度。如果用户输入的数值超过数组的容量，函数GetIntegerArray能报告有错误产生就很重要了。此外，如果GetIntegerArray函数能将输入结束标志值作为参数，那就更通用了，可以调用同样的函数去读入有不同结束标志值的数据串。因此，GetIntegerArray函数原型可以写成：

391

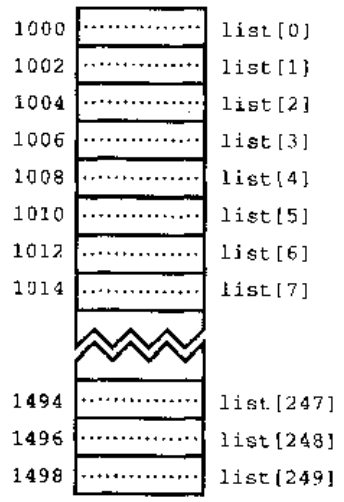
```
int GetIntegerArray (int array[ ],int max,int  sentinel) ;
```

11.3.2 数组参数传递机制

在你实现这些函数之前，必须解决有关数组参数的其他一些问题。当用一个简单变量调用函数时，函数会收到被调用参数的一个拷贝。因为函数是用拷贝进行工作而不是用原来的数值工作，所以函数不会改变调用参数的数值。然而，当一个有数组参数的函数被调用时，形式参数和实际参数之间的关系就会改变。

为了理解数组如何从一个函数传送到另外一个函数，需要考虑数组在内存中的表示方式。假设MaxElements被定义成250，以下的数组声明

```
int list[MaxElements];
```



要求编译器在main的帧中为有250个整型数据的数组分配空间。虽然不可能确切知道在计算机中使用什么地址，我们可以假设分配给数组list的内存起始地址是1000，连续分配250个整型大小的字。如果在你使用的计算机中，整型数据占两个字节，那么数组list的数据将会存储在地址1000～1499中，如右图所示。当你编写一个有数组参数的函数时，通常希望函数直接利用对应于数组的内存中的数值。C语言很好的做到了这一点。当数组作为参数传送给函数时，只有数组的基地址记录在局部帧中。如果你选择在函数范围内定义的局部数组中的一个元素，那么把偏移量加上基地址就会生成在调用数组范围之内的地址。最终结果是在函数头部定义的形式数组参数成为函数调用中实际数组参数的同义词，而不是它的拷贝。

为理解数组操作的细节，可以思考主程序在调用ReverseIntegerArray函数时会有什么情况发生。如果数组含有五个数值1、2、3、4和5，那么对于函数ReverseIntegerArray，帧的初始配置如下图所示：

因为在C语言中，数组参数的行为不同于其他的参数，所以有必要按照下面的规则考虑它们的区别：

> **数组参数规则** 如果调用一个以数组作为形式参数的函数，那么用作实际参数的数组的存储空间被形式参数所共享。改变形式参数中某一元素的值也会改变实际参数数组中的对应元素。

11.3.3 实现函数PrintIntegerArray和GetIntegerArray

为了完成程序reverse.c，需要实现被主程序调用的每一个函数。虽然可以以任何顺序写这些函数，但一般是先写最简单的。在这个例子中，最简单的函数当然是PrintIntegerArray。这个函数的主要任务是遍历并分行输出数组元素。遍历过程只需要一个for循环语句，开始为0，直到遍历完所有的有效元素为止。for循环语句的上限是数组的有效长度n。函数PrintIntegerArray的实现如下：

394

```
static void PrintIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("%d\n", array[i]);
    }
}
```

函数GetIntegerArray的实现有一些复杂，主要是因为你要进行检测以确保用户输入的数据没有超过数组的最大容量。因为C语言没有提供措施避免输入数据超过数组的限制，所以这种检测就至关重要。如果你没有检测且用户输入过多的数据，这些数据就会覆盖其他的重要的信息，使你的程序莫名其妙地不能运行。下面的GetIntegerArray函数的实现就注意检测是否输入过多数据：

```
static int GetIntegerArray(int array[], int max, int sentinel)
{
    int n, value;

    n = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetInteger();
        if (value == sentinel) break;
        if (n == max) Error("Too many input items for array");
        array[n] = value;
        n++;
    }
    return (n);
}
```

一旦实现了GetIntegerArray和PrintIntegerArray两个函数之后，就要在编写ReverseIntegerArray之前对它们进行测试。例如，你可以用下面的主程序对两个函数进行编译：

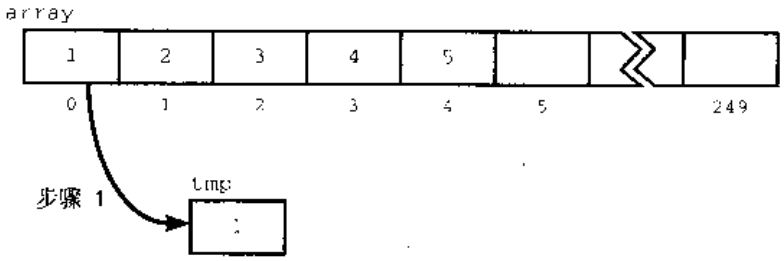
```
main()
{
    int list[MaxElements], n;
```

11.3.5 实现函数SwapIntegerElements

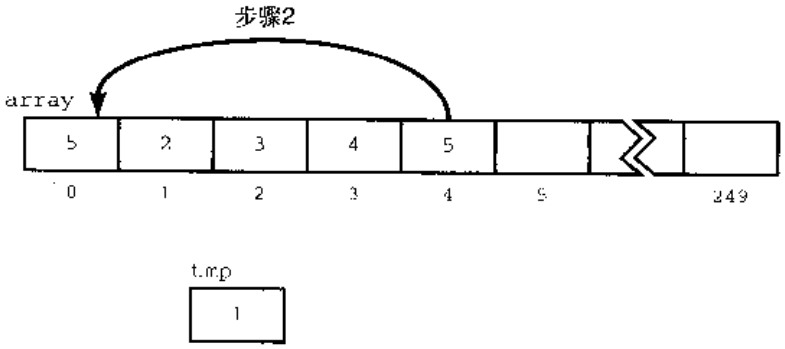
实现函数SwapIntegerElements要比它刚出现的时候复杂一些。你不能把一个元素值赋值给另外一个元素，因为目标元素的初始值会丢失。解决此问题的最简单的办法就是用一个局部变量暂时把此元素值存放起来。如果你保持一个元素值不丢失，你就可以很轻松地对它进行赋值操作了，然后再把暂时存放在局部变量的数值拷贝过来。

假设你想交换下标值为0和4的数组元素。可以分成三步实现这种操作：

1) 把array[0]的值暂时存放到一个临时变量中，如下图所示：

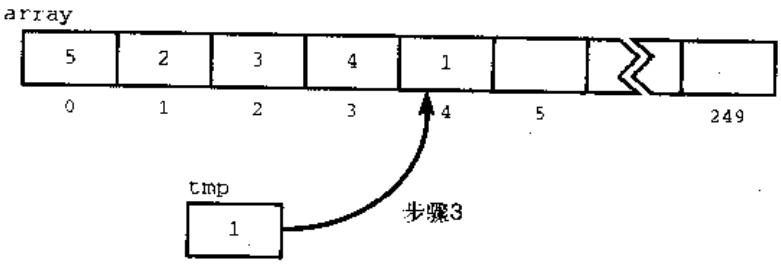


2) 把array[4]中的数值复制到array[0]中，如下图所示：



因为元素array[0]原来的数值已经存放在tmp中，所以没有信息丢失。 397

3) 把tmp中的值赋给array[4]，如下图所示：



这三步是实现函数SwapIntegerElements的基本步骤：

```
static void SwapIntegerElements(int array[], int p1, int p2)
{
    int tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}
```

这个函数补全了reverse.c程序，完整的程序出现如图11-3所示。

```
 * elements, GetIntegerArray generates an error.
 */

static int GetIntegerArray(int array[], int max, int sentinel)
{
    int n, value;

    n = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetInteger();
        if (value == sentinel) break;
        if (n == max) Error("Too many input items for array");
        array[n] = value;
        n++;
    }
    return (n);
}

/*
 * Function: PrintIntegerArray
 * Usage: PrintIntegerArray(array, n);
 * -----------------------------------
 * This function displays the first n values in array,
 * one per line, on the console.
 */

static void PrintIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("%d\n", array[i]);
    }
}


/*
 * Function: ReverseIntegerArray
 * Usage: ReverseIntegerArray(array, n);
 * -------------------------------------
 * This function reverses the elements of array, which has n as
 * its effective size.  The procedure operates by going through
 * the first half of the array and swapping each element with
 * its counterpart at the end of the array.
 */

static void ReverseIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n / 2; i++) {
        SwapIntegerElements(array, i, n - i - 1);
    }
}
```

图 11-3 (续)

入一个空行表示输入结束的时候，程序应该输出一个表，说明在输入数据中各个字母出现的次数。程序的执行可以用如下运行示例说明：

```
This program counts letter frequencies.
Enter a blank line to signal end of input.
Peter Piper picked a peck.↵
of pickled peppers.↵
↵
A     1
C     3
D     2
E     8
F     1
I     3
K     3
L     1
O     1
P     9
R     3
S     1
T     1
```

为了生成上述的字母出现频率表，程序必须逐个字符地搜索输入文本中的每一行。一旦一个字母出现，程序就会更新一个记录此字母出现次数的数值。这个程序值得注意的地方就是要设计一种数据结构，可以记录26个字母出现的次数。

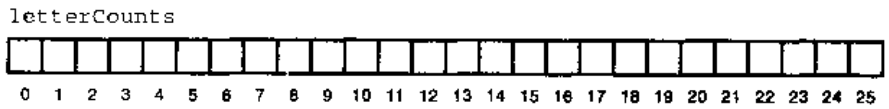
不使用数组，定义26个不同的变量nA，nB，nC依次到nZ也可以解决这个问题，然后用一个switch语句来检查26种情况：

```
switch (toupper(ch)) {
  case 'A': nA++; break;
  case 'B': nB++; break;
  case 'C': nC++; break;
  . . .
  case 'Z': nZ++; break;
}
```

这是一个很长的且重复性的程序。更好的方法就是把26个变量放入一个数组，然后使用字符代码选择数组中合适的元素。每个元素里存放一个整型数值，表示对应于数组中该下标的字母当前出现的次数。如果你调用数组letterCounts，你可以把它声明为： 402

```
int letterCounts[NLetters];
```

Nletters被定义为常数26。这个声明给一个有26个元素的整型数组分配内存空间，如下图所示：



在输入数据中每出现一个字母时，就要增加letteCounts中相应元素的值。寻找需要自增的元素的过程就是用字符运算把一个字母转换成在范围0～25中的整数。这个转换过程由函数LetterIndex完成，该函数的实现如下：

```
int LetterIndex(char ch)
{
    if (isalpha(ch)) {
```

注意，for循环语句的循环范围是从'A'～'Z'，而不是从0～25是很重要的。因为我们用对应于我们问题的下标值，会有利于我们理解for循环语句。同时还应该注意到，在这个例子中也无需检测函数LetterIndex的返回值是否是－1，因为函数DisplayLetterCounts的结构已经确保不会有超出范围的情况发生。

图11-4中程序的其他部分包括逐字扫描输入数据的代码。

```
/*
 * File: countlet.c
 * ----------------
 * This program counts the occurrences of individual letters
 * that appear in text read in from the user.  This program
 * might be useful as a tool in solving cryptograms.
 */

#include <stdio.h>
#include <ctype.h>
#include "simpio.h"
#include "strlib.h"
#include "genlib.h"

/*
 * Constants
 * ---------
 * MaxLines -- Maximum number of input lines
 * NLetters -- Number of letters
 */

#define MaxLines 100
#define NLetters  26

/* Private function declarations */

static void CountLetters(int letterCounts[]);
static void CountLettersInString(string str, int letterCounts[]);
static void RecordLetter(char ch, int letterCounts[]);
static void DisplayLetterCounts(const int letterCounts[]);
static int LetterIndex(char ch);
static void ClearIntegerArray(int array[], int n);

/* Main program */

main()
{
    int letterCounts[NLetters];

    printf("This program counts letter frequencies.\n");
    printf("Enter a blank line to signal end of input.\n");
    ClearIntegerArray(letterCounts, NLetters);
    CountLetters(letterCounts);
    DisplayLetterCounts(letterCounts);
}

/*
 * Function: CountLetters
```

图11-4 countlet.c

```
 * ----------------------------------------
 * This function displays the letter frequency table, leaving
 * out any letters that did not occur.
 */

void DisplayLetterCounts(const int letterCounts[])
{
    char ch;
    int num;

    for (ch = 'A'; ch <= 'Z'; ch++) {
        num = letterCounts[LetterIndex(ch)];
        if (num != 0) printf("%c  %4d\n", ch, num);
    }
}

/*
 * Function: LetterIndex
 * Usage: index = LetterIndex(ch);
 * ------------------------------
 * This function converts a character into the appropriate index
 * for use with the letterCounts array.  In this implementation,
 * LetterIndex converts characters in either case to an integer
 * in the range 0 to 25.  If ch is not a valid letter, LetterIndex
 * returns -1.  Clients should check for a -1 return value unless
 * they are able to guarantee that the argument is a letter.
 */

int LetterIndex(char ch)
{
    if (isalpha(ch)) {
        return (toupper(ch) - 'A');
    } else {
        return (-1);
    }
}

/*
 * Function: ClearIntegerArray
 * Usage: ClearIntegerArray(array, n);
 * -----------------------------------
 * This function sets the first n elements in the array to 0.
 */

static void ClearIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        array[i] = 0;
    }
}
```

图 11-4 (续)

11.5 数组的静态初始化

数组变量可以和其他变量一样被声明为局部变量或全局变量。为了避免使用全局变量的

对于一个数组a，可以通过下面的表达式计算其中的元素的个数：

```
sizeof a/ sizeof a[0]
```

用自然语言描述，这个表达式表示用数组中的第一个元素的大小去除整个数组的大小。因为数组中的每个元素的大小都是一样的，不管元素是什么类型，上述表达式的结果就是数组中元素的个数。因此，你可以用下列表达式初始化一个变量bigCities保存数组中的城市数：

```
static int nBigCities-sizeof bigCities/ sizeof bigCities[0];
```

11.5.3 初始化数组和标量类型

根据第9章中的定义，标量类型可以用在任何整型数据出现的地方。因此可以使用任何标量类型值作为数组的下标值。这样就增加了标量类型的功能，也简化了它们的应用。

作为一个实例，初始化的数组可以很容易把枚举类型的数据转化成与它们相对应的名字，这种操作在C语言中是不直接支持的。例如，如果你想用FALSE和TRUE两个名字来输出布尔类型的数据，你可以声明数组booleanNames如下：

```
static string booleanNames = {"FALSE", "TRUE"};
```

409

然后使用printf语句：

```
printf("flag-%s\n', booleanNames[flag]);
```

因为布尔类型在genlib.b中被定义成：

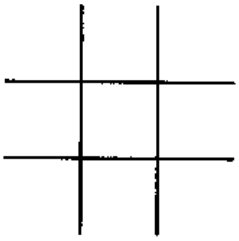
```
typedef enum {FALSE, TRUE} bool;
```

常量FALSE的内部值为0，TRUE的内部值为1。使用这些数值作为数组booleanNames的下标值就可以生成对应于布尔型值的字符串。这种情况也同样适用于连续枚举类型。

11.6 多维数组

在C语言中，数组元素可以是任何类型的。特别是，数组的元素又是数组。这种数组的数组就叫做**多维数组**（multidimensional array）。多维数组中最常见的就是二维数组，它常用于表示分割成行和列的矩形的数据结构。这种二维结构的类型也叫做**矩阵**（matrix）。三维或是多维的数组在C语言中虽然是合法的，但是很少出现。

作为二维数组的例子，假设要在程序中表述tic-tac-toe这种游戏。你可能知道，tic-tac-toe这种游戏是在一个分成三行三列的面板上进行的，如右图所示。参加游戏者轮流在这些空的方块中填入X和O，尽量使相同的符号在水平方向、垂直方向或者对角线上排成一行。



为了描述这种游戏的面板，最好的选择就是使用一个三行三列的二维数组。虽然也可以使用枚举类型来表示在方块中可能出现的三种情况——空格，X和O，但是用char型数据作为元素类型并用字符' '，'X'和'O'表示方块的三种情况可能会更简单一些。这种游戏面板的声明可以写为：

```
char board[3][3];
```

有了上述声明，你就可以通过给出两个下标值来确定面板方块中的字符，一个下标确定行号，一个下标确定列号。在这种表示方法中，行号和列号的范围是0～2，面板中各个方块有如下名字：

410

```
static void DisplayBoard(char board[ ][3])
```

程序可以以同样的方式工作，因为对C语言来讲，知道数组的基地址和第二个下标的大小就可以确定数组board中每一个元素的地址，而不管行数是多少。然而，略去第一个下标值使声明很不对称，因此，一般情况下，在声明多维数组参数时，每一个下标的范围都应该写上。

11.6.2 初始化多维数组

和初始化一维数组一样，可以对多维数组进行静态初始化。为了强调多维数组的整体结构，用来初始化内部数组的数值都被放到一组花括号中。例如：

```
static double identityMatrix[3][3] = {
    { 1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 }
};
```

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

412

声明了一个3×3的浮点数矩阵，并用右图的数据对它进行初始化。

这个特殊的矩阵在数学中经常出现，被称为**单位矩阵**（identity matrix）。

和参数的情况一样，静态初始化的多维数组的声明必须指定所有下标值的范围，第一个下标值除外，因为它可以通过计算初始值个数确定。但如果作为传送参数，最好在声明多维数组时明确指定每个下标值的范围。

关于对多维数组进行静态初始化的更多的实例将在第12章中介绍。

小结

本章介绍了两个截然不同但又相关的概念：数据在内存中的表示和称之为数组的特殊数据结构。有关内存和数据表示，要掌握的基本知识有：

- 每个变量都存储在内存中的某个地方，因此都有一个地址。
- 一旦被分配，变量的地址不再改变，而它的内容可以改变。
- 根据所存储的数据类型，变量需要不同大小的内存。同时随着计算机的不同，相同的变量类型需要的存储空间也有所不同。在某个计算机上编译过的程序中，可以用运算符sizeof来确定存储某一变量值需要的字节数。

对于数组来说，本章讨论以下几项重要内容：

- 数组通常用来存储具有同一数据类型并且按顺序排列的一组数据。数组的每个成员称为*元素*，通常用下标值来标识。在C语言中，所有数组的下标都是从0开始的。
- 当声明一个数组变量时，必须指定数组的大小，而且它必须是一个常量。对于数据量可变的应用来说，常用的方法是在数组中分配足够的空间，使之能保存最大的数据量，而使用时只使用其中的一部分。元素的最大个数称为数组的*分配长度*，实际使用的元素个数称为*有效长度*。 413
- 引用数组中的元素的过程称为*选择*（selection），一般用数组名后跟一个括在方括号内的表达式表示。
- 在内存中，数组元素通常是连续存储的。数组第一个元素的地址称为*基地址*。选择特定元素所需的调整量称为该元素的*偏移量*。
- 当把一个数组作为参数传递给函数时，对应的形式参数被初始化以保存调用数组的基地址。在函数内部，所有涉及到对形式参数进行的操作都转化为对该数组进行操作，因此数组和函数的参数实质上是等同的，而其他的参数类型只是被调参数的拷贝。

17. 本章介绍的确定一个静态初始化数组的元素个数的习语是什么？ 415

18. 在第9章对枚举类型的讨论中，类型directionT的定义如下：

```
typedef enum {North, East, South, West} directionT;
```

给定一个directionT型变量dir，如何用静态初始化数组将方向的名字显示在屏幕上？

19. 什么是多维数组？

20. 假如某数组的基地址是1000，int类型的值在内存中占2个字节，请用图表示下面数组声明中各元素的地址：

```
int rectangular [2][3];
```

21. 为表示8×8的棋盘中每一个地方对应什么棋子，我们将定义什么变量呢？假设该棋盘中每个地方对应的元素是以下元素中的一个：

K	white king	k	black king
Q	white queen	q	black queen
R	white rook	r	black rook
B	white bishop	b	black bishop
N	white knight	n	black knight
P	white pawn	p	black pawn
-	empty square		

如何静态初始化这个数组，使它保存标准的开始位置。

r	n	b	q	k	b	n	r
p	p	p	p	p	p	p	p
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

程序设计练习

1. 写一个程序，用运算符sizeof显示存储各基本数据类型所需要的字节数。在我用过的某台机器上，程序执行的结果可能如下面运行示例所示： 416

```
Values of type char require 1 byte.
Values of type int require 2 bytes.
Values of type short require 2 bytes.
Values of type long require 4 bytes.
Values of type float require 4 bytes.
Values of type double require 12 bytes.
```

因为随着计算机的不同，各数据类型对应的存储空间大小会有所不同，因此输出的结果也会有所不同。

2. 由于各裁判打分时存在某些主观因素，因此在计算平均值时通常去掉一个最高分，去掉一个最低分。写一个程序，读入7个裁判所打的分数，去掉一个最低分，去掉一个最高分，求剩余5个元素的平均值。

3. 在统计学中，一组数据通常有一个**分布**（distribution）。统计分析的主要的目的就是找到一种方法将整个数据集压缩为统计特征，它表示了总体特征。最常见的统计量就是**均值**（mean），也就是简单的平均。对于序列$x_1, x_2, \cdots, x_n$，通常用$\bar{x}$表示其均值。

编写一个函数Mean(array, n)，返回有效长度为n的double型数组的平均值。将该函数加入到图11-1中的gymjudge.c中，测试其是否正确。

4. 另外一个常见的统计量是**标准差**（standard deviation），它表示数列中各值偏离平均值的程度。为计算$x_1, x_2, \cdots, x_n$的标准差，基本步骤如下：

A	•—	J	•———	S	•••
B	—•••	K	—•—	T	—
C	—•—•	L	•—••	U	••—
D	—••	M	——	V	•••—
E	•	N	—•	W	•——
F	••—•	O	———	X	—••—
G	——•	P	•——•	Y	—•——
H	••••	Q	——•—	Z	——••
I	••	R	•—•		

我们很容易在程序中声明一个含有26个元素的数组，存储26个字母的Morse码。

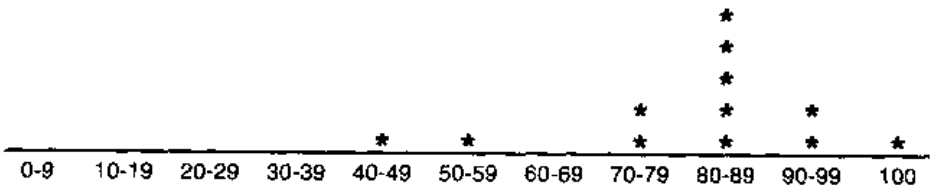
写一个程序，从用户处读入一个字符串，将字符串中每个字母转换成相应的Morse码，用句号表示点，用破折号表示横。用换行符来表示单词之间的空格，而忽略其中的其他标点符号。你的程序应该产生下列运行示例：

```
This program translates a line into Morse code.
Enter English text: What hath God wrought↵
.-- .... .- -
.... .- - ....
--. --- -..
.-- .-. --- ..- --. .... -
```

7. 柱状图是一种显示每个区间内出现的数据的频率的图，例如，对下面的一组成绩：

100，95，47，88，86，92，75，89，81，70，55，80 419

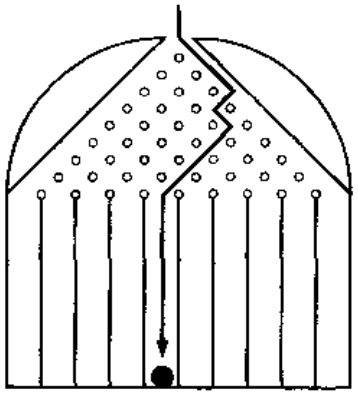
我们可以得到下述的柱状图：



柱状图中的星形表示有一个数据在40～50之间，一个数据在50～60之间，5个数据在80～90之间，依此类推。

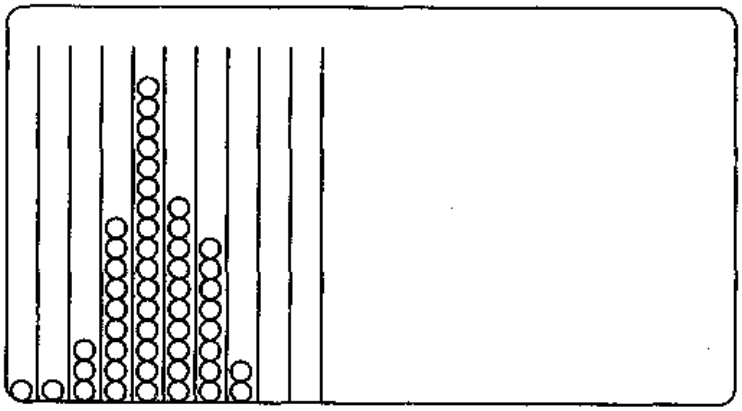
我们用计算机产生柱状图时，显示成以下形式比较容易：

```
0-9   |
10-19 |
20-29 |
30-39 |
40-49 | *
50-59 | *
60-69 |
70-79 | **
80-89 | *****
90-99 | **
100   | *
```



写一个程序，模拟丢下50个弹球，最下面有10个沟道的情况，如下图所示。你编写的程序可以用第7章中的图形库显示图形化的结果。当每一个弹球落地时，程序应在相应的地方画一个表示弹球的图。

下图是50个弹球落下后的一种可能情形：



421
~
422

注意，弹球可能会集中在中间一个沟道中，原因是弹球有更多可能的路径到达中间的沟道。例如要到达最左边的沟道，那么弹球9次都必须从销的左边掉下。相比之下，弹球有很多种路径从顶部到达中间两个沟道，因为左右掉下的顺序可以发生改变而最终它还会到达中间的两个沟道。一般来讲，随机过程最终终止于某个特定状态的可能性取决于到达这个状态的路径数。

11. 用我们在11.6节中给出的tic-tac-toe画板的声明，写一个谓词函数`IsWinningPosition (board, player)`，如果任一个游戏者不管是以`'X'`还是以`'O'`赢得游戏，那么返回`TRUE`。赢的位置是指水平方向、垂直方向或者对角方向的三个元素具有同一字符。编写一个主程序测试函数的功能，读入当前面板数组的内容，然后报告哪个游戏者获得胜利，如下例所示：

```
This program tests the IsWinningPosition function.
Enter the state of the board, row by row.

XOX↵
-XO↵
X-O↵

X has won
```

第12章 查找和排序

Had I been present at the creation, I would have given some useful hints for the better ordering of the universe.

— Alfonso the Wise, thirteenth century

学习目标

- 能够用线性查找和二分查找在数组中找到相应的数据。
- 了解各种算法在效率上的差异。
- 了解一种称为选择排序的简单的排序算法的实现。
- 认识到选择排序算法的效率取决于数据项的个数。

在第11章中，你已经学习了基本的数组操作并且了解了数组在各种应用里如何使用。但是，在第11章中有两个重要的数组操作没有介绍，而这两个重要的操作将在本章中介绍。这两个基本操作就是：

- **查找** 从数组中找出特定元素的过程。
- **排序** 重新排列数组的元素使它们按某一事先确定的顺序存储的过程。

由于查找和排序与数组关系密切，因此从某种意义上说，本章是对数组讨论的继续。但是本章还有另外一个主题，这个主题不仅和第11章有关系，同时和第6章讨论的算法也有关系。由于查找和排序有各种不同的方法，而这些方法的效率又存在很大差异，从而出现了很多有趣的算法问题。

12.1 查找

在前面几章中，我们已经遇到过执行查找的函数，虽然这些函数没有出现在介绍数组的内容中。例如，在piglatin.c程序中（参见图10-5）出现的函数FindFirstVowel的功能就是在一个字符串中查找第一个元音字母。该函数的实现为：

```
int FindFirstVowel(string word)
{
    int i;

    for (i = 0; i < StringLength(word); i++) {
        if (IsVowel(IthChar(word, i))) return (i);
    }
    return (-1);
}
```

该实现采用简单直接的算法来解决查找问题。从字符串的第一个字母开始，该函数依次检查每一个字母，如果该字母是元音字母，那么FindFirstVowel返回该字母在原字符串中的下标。如果在整个字符串中都没有发现元音字母，那么FindFirstVowel将返回值－1告诉客户没有元音字母出现。

12.1.1 在整数数组中查找

你可以利用以上方法在一个数组中查找某个数据项。例如，函数FindIntegerInArray

```
/* Main program */

main()
{
    int value, index;

    printf("This program looks up names of U.S. coins.\n");
    printf("Enter coin value: ");
    value = GetInteger();
    index = FindIntegerInArray(value, coinValues, nCoins);
    if (index == -1) {
        printf("There is no such coin.\n");
    } else {
        printf("That's called a %s.\n", coinNames[index]);
    }
}

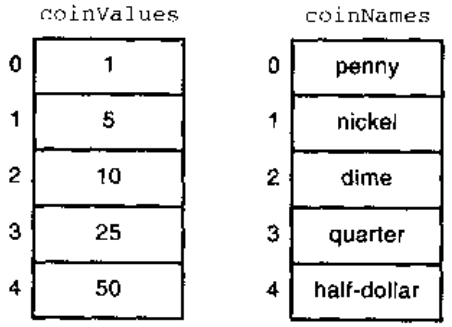
/*
 * Function: FindIntegerInArray
 * Usage: index = FindIntegerInArray(key, array, n);
 * ------------------------------------------------
 * This function returns the index of the first element in the
 * specified array of integers that matches the value key.  If
 * key does not appear in the first n elements of the array,
 * FindIntegerInArray returns -1.
 */

static int FindIntegerInArray(int key, int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        if (key == array[i]) return (i);
    }
    return (-1);
}
```

图 12-1 (续)

FindIntegerInArray函数返回数组coinValues中相应硬币值对应的下标。如果硬币是美国铸造的五种标准硬币之一的话，返回的下标再用来在coinNames数组中选择相应的元素，也就是得到硬币的名称。这两个数组是通过以下原则联系在一起的：硬币的价值的下标和对应的硬币名称的下标是相同的，如右图所示。用相同的下标位置来存储相关的数据称为**并联数组**（parallel array）。



428

12.1.2 关于查找的另一个更复杂的例子

在开始讨论不同查找算法的时候，我们先介绍一个更复杂的查找的应用，而它将就刚才提到的问题做出更容易理解的描述。假设你要在程序中表示一张表（如下所示），该表给出各城市之间的距离[⊖]（用英里表示）：

⊖ 数据来源：*Rand McNally Road Atlas 1994*, New York: Rand McNally。

```
    "San Francisco",
    "Seattle",
};
```

现在已经有了数据，接下来要考虑的就是如何写一个程序读入两个城市的名字，显示它们之间的距离，运行结果如下所示：

```
This program looks up intercity mileage.
Enter name of city #1: San Francisco↵
Enter name of city #2: Boston↵
Distance between San Francisco and Boston: 3128 miles.
```

430

程序必须执行以下步骤：

1) 以字符串的形式读入两个城市的名字。

2) 在数组cityTable中找到该城市名字所对应的下标位置。

3) 用下标值在数组mileageTable中找到对应的值，这个值就是两城市之间的距离。

假设你能够实现一个在字符串数组中查找的函数FindStringInArray，那么程序的其余部分可以直接写成：

```
main()
{
    int city1, city2;

    printf("This program looks up intercity mileage.\n");
    city1 = GetCity("Enter name of city #1: ");
    city2 = GetCity("Enter name of city #2: ");
    printf("Distance between %s", cityTable[city1]);
    printf(" and %s:", cityTable[city2]);
    printf(" %d miles.\n", mileageTable[city1][city2]);
}

static int GetCity(string prompt)
{
    string cityName;
    int index;

    while (TRUE) {
        printf("%s", prompt);
        cityName = GetLine();
        index = FindStringInArray(cityName, cityTable, NCities);
        if (index >= 0) break;
        printf("Unknown city name -- try again.\n");
    }
    return (index);
}
```

12.1.3 线性查找

以上程序中唯一缺少的是函数FindStringInArray的代码。如果遵循实现FindIntegerInArray的逻辑，那么只需要改变参数类型，在循环中使用StringEqual比较字符串的值就可以了。相应的代码和注释见图12-2。

San Francisco所在位置一定是在Houston的后面，因为这个数组是按照字母顺序排列的。因此 433
你可以立即排除下标值从0～5的城市名，留下下列这些位置：

cityTable

0	~~Atlanta~~
1	~~Boston~~
2	~~Chicago~~
3	~~Denver~~
4	~~Detroit~~
5	~~Houston~~
6	Los Angeles
7	Miami
8	New York
9	Philadelphia
10	San Francisco
11	Seattle

仅仅通过上面的一步操作，你已经删去了表中一半的城市名。事实上，更重要的是，在下面的查找中你可以采用和上面相同的方法。如果San Francisco在CityTable中的话，那么它的位置一定是在6～11之间。用上面提到的方法我们再计算该范围的中心：

$$\frac{6+11}{2}$$

由于是整数运算，因此上面表达式的值是8。而同样由字母顺序可知，San Francisco一定在New York的后面，那么又排除了三个城市。

cityTable

0	~~Atlanta~~
1	~~Boston~~
2	~~Chicago~~
3	~~Denver~~
4	~~Detroit~~
5	~~Houston~~
6	~~Los Angeles~~
7	~~Miami~~
8	~~New York~~
9	Philadelphia
10	San Francisco
11	Seattle

434

当你在剩下的范围中再找中间位置时，你发现San Francisco的下标是11，因此整个搜索过程只需要三步。

首先在已排序的数组中查找中间元素，然后根据这个元素的值确定下一步将在哪个部分

```
 * range and compares the element at that index position to the
 * key.  If there is a match, the function returns the index.
 * If the key is less than the string at that index position, the
 * function searches in the first half of the array; if the key is
 * larger, the function searches in the second half of the array.
 */

int FindStringInSortedArray(string key,
                            string array[],
                            int n)
{
    int lh, rh, mid, cmp;

    lh = 0;
    rh = n - 1;
    while (lh <= rh) {
        mid = (lh + rh) / 2;
        cmp = StringCompare(key, array[mid]);
        if (cmp == 0) return (mid);
        if (cmp < 0) {
            rh = mid - 1;
        } else {
            lh = mid + 1;
        }
    }
    return (-1);
}
```

图 12-3（续）

因此，使用线性查找算法查找由N个元素组成的数组需要N次的比较，而使用二分查找算法只需要$\log_2 N$次比较就可以了。

用数学形式表示某些算法的相对效率是定量评价这些算法效率的非常有用的手段。然而对大多数人来说，这种公式仍然不能给人们一个直观的感觉，到底哪个算法更好。那么，我们可以看一些数据。右边的表给出了不同的N值和它相对应的最精确的$\log_2 N$的整数值。从表中的数据可以看出，对于小的数组，两种算法都能很好地完成搜索任务。然而，如果数组的元素个数为1 000 000 000时，在最坏的情况下线性查找算法需要1 000 000 000次的比较才能查找完毕；而二分查找算法最多只需要30次的比较就能查找完毕。很明显，比较次数的锐减表示算法效率大幅增加。

436

N	$\log_2 N$
10	3
100	7
1000	10
1 000 000	20
1 000 000 000	30

二分查找算法唯一的问题就是数组必须是严格地按照某个顺序排列的。如果不是，则必须采用线性查找算法。或者自己先把该数组按要求进行排序。对数组进行排序是比查找更具挑战性的工作，接下来的几节中将介绍这方面的知识。

12.2 排序

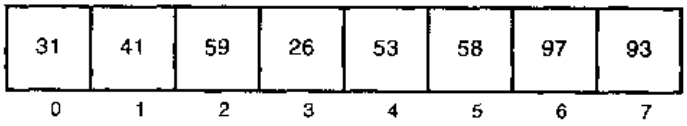
在大多数的商业应用中，计算机通常用来处理一些很简单的工作，例如求某一个数据序列的数值之和或者求它们的平均值，这正好是我们在前几章中已经解决的问题。然而，在某些商业性的程序设计过程中，所要用到的重要操作要更复杂一些。在这些复杂的操作中，最重要的就是**排序**（sorting），它是把一个表（通常表示为数组）按某一次序重新排列。例如，

而实现相应的sort.c程序并不像想象中的那么简单，尤其是当你想找到一种高效率的方法对数据进行排序的时候，情况更是如此。与计算机科学领域内出现的很多问题一样，这里有很多可用的算法。在高级计算机科学教材中，可能需要好几周的时间学习各种不同的排序方法，而每一种算法都有各自的优点和缺点。而在目前的学习阶段，最好从一个排序算法开始，把它的每一个细节都理解清楚。

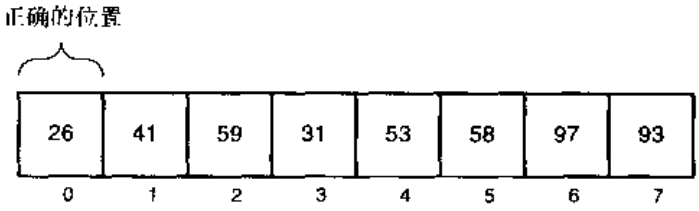
12.2.2 选择排序算法

在众多的排序算法中，最容易理解的一种算法就是**选择排序**（selection sort）算法。当应用选择排序算法时，每次将其中的一个元素放在它最终要放的位置。第一步是找到整个数组中的最小的元素并把它放在它最终要放的位置上，即数组的起始位置，第二步是在剩下的元素中找最小的元素并把它放在第二个位置上。对整个数组重复这个过程，最终将得到按从小到大顺序排列的数组。

为了理解选择排序算法，我们以下面这个数组作为例子：



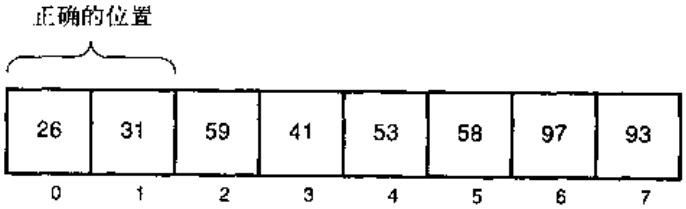
由于该组数中最小的元素值是26，它在数组中的位置是3，因此需要将它移动到位置0。和图11-3中的reverse.c程序一样，我们不希望丢失位置0中的原始值，因此最简单的办法是交换位置0和位置3的值。经过交换得到的新数组如下：



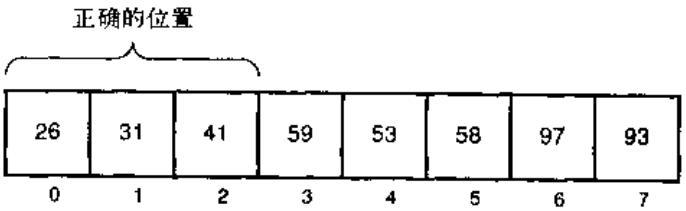
经过交换，位置0的值就是该数组的最小值，符合最终的排序要求。

现在，可以处理表中剩下的数据。下一步是用同样的策略正确填入数组的第2个位置中的值。最小的值（除了已经被正确放置的26外）是31，现在它的位置是3。如果将它的值和下标位置为1的元素的值进行交换，可以得到下面的状态，前两个元素是正确的值：

438 ~ 439



在下一个周期中，再将下一个最小值（应该是41）和位置2中的元素值进行交换：



的SwapIntegerElements是同一个函数，在此我们就不再赘述。当你写完一个实现某个通用操作的函数时，最好把它放在手边，以便以后使用。有经验的程序员通常尽可能重用已有的代码，这样就可以省去由编写和调试这部分程序带来的麻烦。 441

只需要将SwapIntegerElements函数的代码拷贝到选择排序算法中即可完成整个选择排序算法。sort.c实现的完整代码参见图12-5。

12.2.3 选择排序效率的评估

选择排序算法有很多优点。首先，它的算法很容易理解；其次，它解决了排序这个问题。但是，还存在其他一些更有效的排序算法。但是，效率最高的排序算法需要很高的技巧，超出了你现在所掌握的程序设计知识。由于这个原因，我们将在第17章中再讨论更有效的排序算法。

```
/*
 * File: sort.c
 * ------------
 * This file implements the sort.h interface using the selection
 * sort algorithm.
 */

#include <stdio.h>
#include "genlib.h"
#include "sort.h"

/* Private function prototypes */

static int FindSmallestInteger(int array[], int low, int high);
static void SwapIntegerElements(int array[], int p1, int p2);

/*
 * Function: SortIntegerArray
 * --------------------------
 * This implementation uses an algorithm called selection sort,
 * which can be described in English as follows.  With your left
 * hand, point at each element in the array in turn, starting at
 * index 0.  At each step in the cycle:
 *
 * (1)  Find the smallest element in the range between your left
 *      hand and the end of the array, and point at that element
 *      with your right hand.
 *
 * (2)  Move that element into its correct index position by
 *      switching the elements indicated by your left and right
 *      hands.
 */

void SortIntegerArray(int array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n; lh++) {
        rh = FindSmallestInteger(array, lh, n-1);
        SwapIntegerElements(array, lh, rh);
    }
}
```

图12-5 sort.c

达式将它转化成以秒来计算的时间形式：

```
(double) clock ( ) /CLOCKS_PER_SEC
```

如果把开始时间和结束时间分别存储于变量start和finish中，可以用下列代码计算执行一个操作所需的时间：

```
double start, finish, elapsed;

start = (double) clock ( ) / CLOCKS_PER_SEC;
. . . 执行某些计算 . . .
finish = (double) clock ( ) / CLOCKS_PER_SEC;
elapsed = finish - start
```

你也可以用上述方法计算图12-5中的选择排序算法的运行时间。对大小不同的数组调用SortIntegerArray函数所需要的时间见表12-1。在这张表中，*N*表示数组中元素的个数，而“运行时间”列表示使用选择排序算法对相应的数组进行排序所需要的时间（时间单位为毫秒）。

这张表显示了一个有趣的现象。当*N*很小时，选择排序算法需要的时间很少，但随着*N*的增大，执行选择排序算法需要的时间显著增加。举个例子来说，如果数组包含30个值，SortIntegerArray排序这个数组需要1毫秒，当达到800个值时，需要的时间超过0.5秒。而商业应用通常需要对10 000、100 000甚至更多的数进行排序。对这种规模的数组，选择排序算法的速度将慢得惊人。

442
~
444

表12-1 选择排序算法的运行时间（以毫秒为单位）

N	运行时间	*N*	运行时间
10	0.13	100	9.67
20	0.33	200	37.33
30	1.00	400	146.67
40	1.47	800	596.67
50	2.40		

12.2.5 选择排序的算法分析

要理解为什么程序执行的时间会如表12-1所示，首先我们要明白算法的工作原理。考虑表12-1中选择排序的时间数据，当*N*为50时，执行算法需要2.4毫秒，而当*N*为100时，需要的执行时间为9.67毫秒，差不多是*N*为50时需要时间的4倍。我们从表中其他部分也可以得到这个结论：数组的元素个数增加一倍，那么所需要的时间是原来的4倍。我们把这个排序算法称为具有**平方律**（quadratic）的性质，即执行时间和输入数组大小的平方成正比。

选择排序算法具有平方律性质的事实其实并不奇怪，你只要想一下算法的执行过程就可以理解。在对一个具有8个元素的数组进行排序时，该算法需要执行8次外层for循环。第一个循环周期找出8个数中的最小值，下一个循环周期则在剩下的7个元素中找到最小值，依次类推。程序执行的操作次数和该数组的个数成正比，对上面的例子，需要执行的操作的次数为：

$$8+7+6+5+4+3+2+1$$

更一般的情况，假如数组有*N*个数，选择排序需要的时间与下面的和成正比，即：

$$N+N-1+N-2+\cdots+3+2+1$$

而前*N*个数之和我们可以通过数学公式得到更简洁的表达式，即：

$$N+N-1+N-2+\cdots 3+2+1=\frac{N^2+N}{2}$$

前两个色带的颜色代表阻抗的前两位，第三个色带表示10的幂次数，它要和前两位相乘。例如，当颜色依次是黄-紫-橙，那么它对应的阻抗是：

$$47 \times 10^3$$

即47 000Ω。

447

编写一个程序，读入电阻上三个色带的颜色，输出相应的阻抗值，如下面运行示例所示：

```
This program interprets the resistor color code.
Color of first band: Yellow↵
Color of second band: Violet↵
Color of third band: Orange↵
Resistance = 47000 ohms.
```

2. 编写一个和用户猜数字的程序guestnum.c，假设这些用户是小学的学生。首先由用户确定数字，然后回答计算机的一系列问题，直到最终计算机确定这个数字为止。下面的运行示例显示了当用户确定的数字是17时的执行结果：

```
Think of a number between 1 and 100 and I'll
guess it.
Is it 50? no↵
Is it less than 50? yes↵
Is it 25? no↵
Is it less than 25? yes↵
Is it 12? no↵
Is it less than 12? no↵
Is it 18? no↵
Is it less than 18? yes↵
Is it 15? no↵
Is it less than 15? no↵
Is it 16? no↵
Is it less than 16? no↵
Is it 17? yes↵
I guessed the number!
```

3. 写一个谓词函数IsSorted(array, n)，它的参数是数组名和有效长度，如果数组按照非递减次序排列，返回TRUE。

4. 扩展图12-4和图12-5给出的sort.h和sort.c文件，使其不仅输出SortIntegerArray函数，同时还输出一个函数Alphabetize，它对一个字符串数组按词典顺序排序。

5. 在第11章中，习题第3题和第4题要求编写一个程序来计算两个常见的统计量：平均值和标准差。而另一个常用的统计量就是**中值**（median），也就是把数据按照从小到大的顺序排列后处于中间位置的值。如果数组中元素的个数为偶数，那么就不存在中间元素，此时标准的做法是取中间两个元素的平均值。

448

写一个函数Median(array, n)返回浮点数数组的中值。在写程序时不能假设这个数组是排好序的，因此在程序运行时可能改变元素的次序。

6. 除了中值和平均值，一个分布中最有代表性的第三个统计量是表示数组中出现频率最多的元素的值，例如在数组

9. 很多算法问题在它们的解决方案中都和排序有关。例如，你可以根据随机键值“排序”，以打乱一个数组。一种方法就是使用选择排序算法，然后把找到最小值的位置改成选择一个随机位置。该结果是一个打乱算法，其中各种可能的输出是等概率的。

编写一个程序shuffle.c，通过随机排序输出1～52之间的整数。

10. 入门课程中的一个最著名的算法是所谓的荷兰国旗问题，最初由Edsger Dijkstra提出的。假设一个有n个元素的数组，该数组的每个元素是字符——'R'、'W'或'B'，表示荷兰国旗上的颜色之一。开始时，数组中的值是乱七八糟的，如下所示：

R	B	W	W	B	B	R	W	W	R	R	W	R	B	W
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

你需要做的是重新对以上数组进行排序使其按照荷兰国旗的顺序排序，即所有的红色后面都是白色，而所有白色后面都是蓝色。 450

研究解决这个问题的运行示例，找出相应算法：

```
Initial state:
R B W W B B R W W R R W R B W
Swapping positions 1 and 14
R W W W B B R W W R R W R B B
Swapping positions 4 and 13
R W W W B B R W W R R W R B B
Swapping positions 4 and 12
R W W W R B R W W R R W B B B
Swapping positions 1 and 4
R R W W W B R W W R R W B B B
Swapping positions 5 and 11
R R W W W W R W W R R B B B B
Swapping positions 2 and 6
R R R W W W W W W R R B B B B
Swapping positions 3 and 9
R R R R W W W W W W R B B B B
Swapping positions 4 and 10
R R R R R W W W W W W B B B B
```

编写一个程序对任一随机构造的初始数组实现这个算法。

11. 除了在现阶段比较容易理解的选择排序算法外，还有很多其他的排序算法。但是，这些算法在性能上与选择排序算法相比并没有什么显著优点。即使如此，编写这些算法对我们练习数组的使用是很有帮助的。

例如，你也可以应用以下方法对一个整型数组从小到大进行排序。从数组的起始位置开始，检查数组的每一个元素。对邻近的两个元素进行比较，如果这两个数按照从小到大顺序排列，那么继续向下执行，如果这两个元素不是按照从小到大的顺序排列，那么就将它们的位置进行交换。在任何一种情况下，都移到下一对数据。当在表中移动时，每对数据会有重复。第一对中的第二个元素成为下一对中的第一个元素。重复上述操作直到检查完所有元素，即不再需要交换任意两个元素。这个算法被称为**冒泡排序**（bubble sort），因为元素似乎是“浮到”了它最终的位置。

用上述所说的冒泡法排序重新实现函数SortIntegerArray。

第13章 指　针

> Orlando ran her eyes through it and then, using the first finger of her right hand as pointer, read out the following facts as being most germane to the matter.
>
> —Virginia Woolf, *Orlando*, 1928

学习目标

- 认识到能够使用地址作为数据值的重要性。
- 能够使用指针运算符&和*。
- 能够通过引用调用在函数及其调用程序之间共享数据。
- 认识指针和数组间的关系。
- 详细理解运算符++和--的使用方法。
- 能在程序运行时应用动态分配预留新的存储空间。

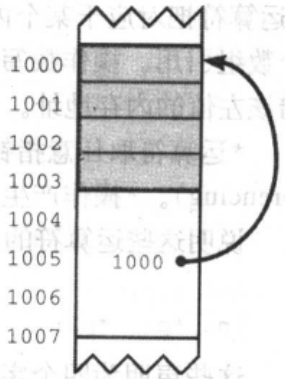
本章介绍**指针**（pointer）的概念，指针是一个数据项，它的值是其他值在内存中的地址。许多高级程序设计语言很少使用指针，因为这些语言提供了其他的机制避免使用指针。C语言的设计意图是让程序员尽可能多地访问由硬件本身提供的功能，因而指针的使用非常普遍。所以，不理解指针的工作原理就不能理解C程序。此外，如果想成为出色的C语言程序员，必须更深入地学习，学习如何在应用中更加有效地使用指针。

在C语言中，指针有多种用途，下面列出的是一些最重要的用途：

- *指针允许你以更简洁的方式引用大的数据结构*　程序中的数据结构可以任意大，但无论如何增长，数据结构总是位于计算机的内存中，因此必然会有地址。利用指针，就可以使用地址作为一个完整值的速记符号，因为一个内存地址在内部表示为一个整数。当数据结构本身很大时，这种策略能节约大量内存空间。本书在以后还会反复提到这一原则，特别是在第14章和第16章中。
- *指针使程序的不同部分能共享数据*　如果将某一个数据值的地址从一个函数传递到另一个函数，这两个函数就能使用同一数据。指针的这一应用将在本章的13.3节进一步解释。
- *利用指针，能在程序执行过程中预留新的内存空间*　到现在为止，在程序中能使用的内存就是通过显式声明分配给变量的内存。在许多应用中，如果程序能在运行时获得新的内存空间，并让指针指向这一内存则更为方便。这种策略将在本章的13.5节继续讨论。
- *指针可用来记录数据项之间的关系*　在高级程序设计应用中，指针被广泛用于构造单个数据值之间的联系。比如，程序员通常在第一个数据的内部表示中包含指向下一个数据项的指针，来说明这两个数据项之间有概念上的顺序关系。虽然对指针应用的完整讨论超出了本书的范围，但此内容在第17章中还会进行简要介绍。

了解了大多数程序设计的概念以后，掌握指针的使用要求你同时从归约和整体两方面考

程序员常在内存图上画上箭头，如右图所示。当然，在计算机内部是没有箭头的。地址1004里的字只是包含了与数值1000对应的位串。同样的位串作为整数使用还是作为地址使用取决于变量在程序中是怎样声明的。如果将变量声明为指针，就可以把存放在位置1004的值1000理解成内存中变量i的地址，并使用指针检索或操作i的值。



13.2 C语言的指针操作

C语言具有操作指针的能力，就如具有操作整数的能力一样。和任何其他数据类型一样，你可以存储指针变量中的地址值，或是将地址值作为参数传递给函数。但在此之前，考虑一些C语言中控制指针使用的细节是很重要的。

13.2.1 在C语言中声明指针变量

和C语言中其他变量一样，使用指针变量前必须先对它们进行声明。为了声明一个指针变量，需要使用下面框中的声明句法。比如，下面的声明

```
int *p;
```

将变量p声明为概念上指向整型的类型。类似的，声明

```
char *cptr;
```

表示声明变量cptr为指向字符型的类型，虽然这两种类型在计算机内部都是以地址形式表示的，但指向整型的指针和指向字符型的指针在C语言中还是有区别的。要使用该地址中的数据，编译程序必须知道如何解释地址里的数据，因此就要求显式地说明指针所指向的数据的类型。指针指向的值的类型称为指针的**基本类型**（base type）。所以，指向整型的指针的基本类型为int。

语法：指针声明

base-type ******pointer-variable*;

其中：

*base-type*是指针指向的值的类型。

*pointer-variable*就是正在被声明的变量。

必须注意，表示变量为指针的星号在语法上属于变量名，不属于基本类型。如果使用同一个声明语句来声明两个同类型的指针的话，你必须给每个变量都加上星号标志，如：

```
int *p1, *p2;
```

而声明

```
int *p1, p2;
```

则表示声明p1为指向整型的指针，而p2是整型变量。

常见错误 新程序员有时会忘记在声明指针时在每个变量前都必须加上*。比如，声明

```
char *cp,c;
```

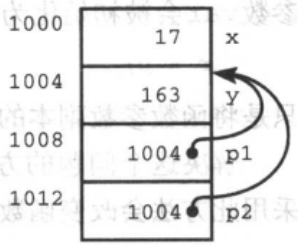
把cp声明成指向char的指针类型，而c是char类型的变量。

13.2.2 基本的指针操作

C语言定义了两种操作指针值的运算符：

- & 取地址。

命令计算机取出变量p2的值并将该值复制给变量p1。p2的值是指针值1004。如果你将这个值赋值给p1，那么p1和p2就会同时指向变量y，如右图所示。就发生在机器内部的操作来说，复制一个指针和复制一个整数是完全一样的。指针的值被原封不动地复制到目的地。从图的概念角度来说，复制指针的效果就是将目的地指针处的箭头替换成与原指针指向同一位置的箭头。所以，赋值语句




```
p1 = p2;
```

的效果就是改变了从p1出发的箭头，使其和从p2出发的箭头指向同一内存。 460

能够区分指针赋值和值赋值是很重要的。形如

```
p1 = p2;
```

的指针赋值使p1和p2指向同一位置。而表示为语句

```
*p1 = *p2;
```

的值赋值是把以p2为地址的内存位置里的值复制到以p1为地址的内存位置里去。

13.2.3 特殊指针NULL

在指针的应用中，特别是在超出本书的范围的应用中，能在一个指针变量中存储一个特殊值表示变量实际上并不指向任何有效数据是很有用的。为了达到这个目的[⊖]，C语言定义了一个特殊的常量NULL。常量NULL可以赋给任何指针变量，在机器内部表示为地址值0。

如果一个指针变量的值为NULL，很重要的一点就是，不要用*运算符来间接引用此变量。NULL值的目的是表示指针不指向有效数据，所以想要找出和NULL指针有关的数据是没有意义的。遗憾的是，大多数编译器没有查找这一错误的程序。如果你间接引用NULL，通常计算机会寻找存放在内存地址0中的值。如果你正巧使用NULL指针赋值来改变那个值的话，程序会崩溃，也不会有任何关于问题如何产生的线索。未初始化的指针变量也会发生同样的问题。

NULL指针的各种使用会在介绍和它们相关的应用时介绍。现在要记住的是，有这样一个常量存在。

常见错误 小心不要间接引用那些没有初始化或值为NULL的指针，这样做会引用不属于本程序的内存空间，很可能会使程序崩溃。

13.3 通过引用传递参数

为了理解指针变量在实际中是如何使用的，让我们看一下C语言中指针的最常见的应用之一，即传递指针给函数，允许该函数操作其调用程序的数据。在C语言中，把一个简单变量从一个函数传递到另一个函数时，该函数就得到了一个调用值的拷贝。在函数的语句中给参数赋一个新值会改变参数在这个函数中的局部副本，但不会影响调用参数。例如，如果要实现 461
一个用以下方法将某个变量初始化为0的函数：

```
void SetToZero(int var)
{
    var = 0;
}
```

（此程序是有错的，事实上无效）

该函数不起任何作用。如果调用

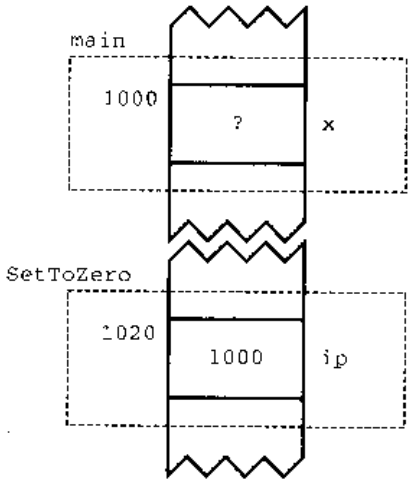
⊖ 常量NULL事实上是在stdlib.h头文件中定义的，当程序包含stdio.h头文件时，就会自动包含这个头文件。因为本书中所有程序都包含stdio.h，所以你可以把NULL当成是C语言内置的常量。

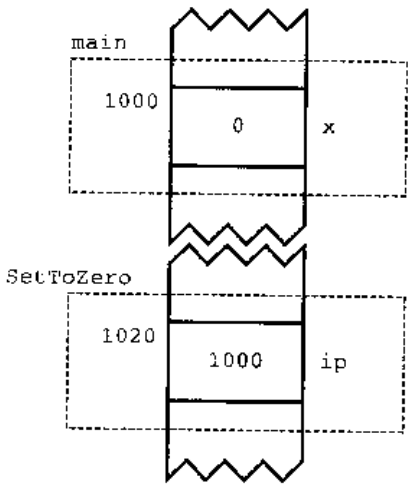
这个帧同样也处于内存的某处，因此内存应该如下面的左图所示[⊖]。

语句

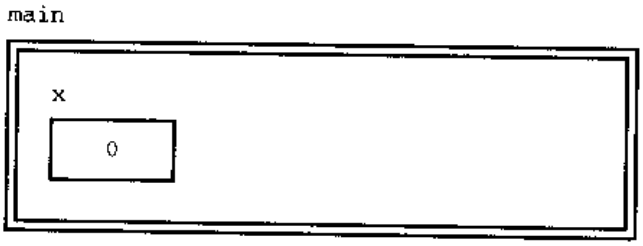
```
*ip = 0;
```

能把地址为ip的整型字的值置为0，如下面的右图所示：





当SetToZero函数返回时，地址1000中值的改变还是有效的，所以main函数的帧现在如下图所示。指针作为参数使函数能够改变其调用函数的帧中的值。在C语言中，你必须通过声明参数值为指针类型明确说明想要改变值，然后将地址作为参数传递。在很多其他语言中，这种机制是语言定义的一部分，称为**引用调用**（call by reference）。虽然C语言的方法并不那么方便，并且需更长的时间来学习，但它确实有自己的优势。其最重要的优势在于函数调用自身的句法指出了参数变量值在函数执行的过程中是否能被改变。例如，在完全不了解函数Mystery的情况下，你却可以知道变量x的值在进行如下调用后没有发生改变：




```
Mystery(x);
```

如果Mystery要改变x的值的话，函数必须重新定义为有一个指针参数，则调用就变为

463
|
464

```
Mystery(&x);
```

应用这一规则使得我们容易预测某个函数调用的结果。

13.3.1 设计函数SwapInteger

为了说明引用调用是怎样影响程序设计的，让我们回到图12-5表示的选择排序算法。SortIntegerArray过程的实现如下：

⊖ 实际上哪个地址会分配给这些帧是没有办法预测的，甚至连这两个帧的相对方位也是无法预测的。在大多数现代的计算机，一般SetToZero帧的内存地址比main帧的内存地址小，但理解内存结构并不依赖于任何这样的假定。

```
/*
 * File: hours.c
 * -------------
 * This program converts a time given in minutes into
 * separate values representing hours and minutes.  The
 * program is written as an illustration of C's mechanism
 * for simulating call by reference.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Constants */

#define MinutesPerHour 60

/* Function prototypes */

static void ConvertTimeToHM(int time, int *pHours, int *pMinutes);

/* Test program */

main()
{
    int time, hours, minutes;

    printf("Test program to convert time values\n");
    printf("Enter a time duration in minutes: ");
    time = GetInteger();
    ConvertTimeToHM(time, &hours, &minutes);
    printf("HH:MM format: %d:%02d\n", hours, minutes);
}

/*
 * Function: ConvertTimeToHM
 * Usage: ConvertTimeToHM(time, &hours, &minutes);
 * -----------------------------------------------
 * This function converts a time value given in minutes into
 * an integral number of hours and the remaining number of minutes.
 * Note that the last two arguments must be passed using their
 * addresses so that the function can correctly set those values.
 */

static void ConvertTimeToHM(int time, int *pHours, int *pMinutes)
{
    *pHours = time / MinutesPerHour;
    *pMinutes = time % MinutesPerHour;
}
```

图13-1 hours.c

在这个实现中，这些值被表示为指向实际变量的指针。所以，为了计算小时数，合适的赋值语句为：

```
*pHours = time/MinutesPerHour;
```

你要用*运算符来保证该结果存储在pHours指向的变量中。

是一个左值，所以书写

```
&list[i]
```

是合法的，它表示list中第i个元素的地址。

由于list中第i个元素的地址取决于变量i的值，所以C语言编译器在编译程序时是无法计算这一地址的。为了确定这一地址，编译器产生一段指令，取数组的基地址，加上适当的偏移量，偏移量是将i的值乘以每个数组元素的字节数而得到的。所以，计算list[i]的地址可以由公式

$$1000 + i \times 8$$

得到。例如，如果i为2，地址计算的结果即为1016，与图中显示的list[2]的地址是一致的。因为计算任何一个数组元素地址的过程是自动的，所以在写程序时无需考虑计算的细节问题。

13.4.1 指针运算

在第2章介绍运算符+和-时，你很可能还未认真地考虑过它们。毕竟，从小学开始就开始使用这两个运算符了。当+和-的操作数为数字时，结果由简单的数学运算决定。在C语言中，469
也可以将这些运算符应用于指针，在某些方面，其结果与数学运算相似，但在另外一些方面又有所不同。对指针值应用加减的过程称为**指针运算**（pointer arithmetic）。

下面的规则对理解指针运算工作机制是很有必要的。

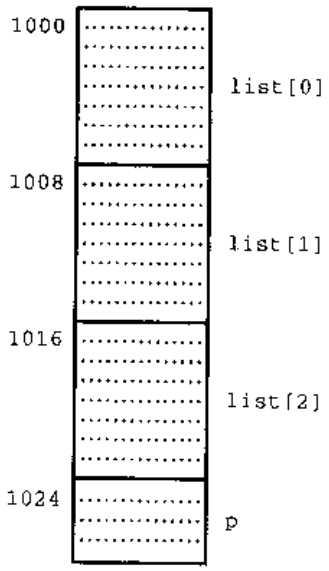
> **指针运算规则** 如果指针p指向数组arr的第一个元素且k为整型数，以下的等式总是成立的：p+k定义为&arr[k]。

换句话说，如果你在指针值上加上一个整数k，则结果就是下标为k的元素的地址。

为了解释指针运算规则，假设有一个函数包含如下所示的变量声明：

```
double list[3];
double *p;
```

在函数帧内给每个变量都分配了空间。对数组变量list，编译器为它的三个元素都分配了足够的空间以便存放一个double型的数据。对指针p，编译器为其分配了足够空间来存放某个double类型的左值的地址。如果帧从地址1000开始，内存分配如右图所示。



由于还未给这些变量赋值，所以无法确定其初始内容。但是可以给它们赋值。例如，可以使用以下的赋值语句将任意值存储到每个数组元素中去：

```
list[0] = 1.0;
list[1] = 1.1;
list[2] = 1.2;
```

要初始化指针变量p，使其指向数组的起始地址，可以执行以下赋值语句：

```
p = &list[0];
```

13.4.2 运算符++和--的新作用

在深入探索指针的神秘之处之前，有必要重新看一下自增和自减运算符，即在第3章中介绍的++和--。到目前为止，本书中的程序仅将这些运算符用于操作数，对它们所操作的左值加1或者减1。

但在实际情况中，这些运算符可以比前面例子中出现的情况更加复杂。首先，这两个运算符都可以用两种方式写，我们熟悉的一种写法是将运算符写在操作数后，即

```
x++
```

或者将运算符写在操作数前，即

```
++x
```

第一种形式，即运算符在操作数之后，称为**后缀**（postfix）式，而第二种形式称为**前缀**（prefix）式。

如果所要做的只是执行单独++运算，就像把它作为一个单独的语句，或作为for循环中的自增运算符那样，前缀运算符和后缀运算符的效果是完全一样的。只有在把这两个运算符作为更长的表达式的一部分时，区别才显现出来。和所有的运算符一样，++运算符会返回一个值，但为何值取决于运算符相对于操作数的位置。这两种情况如下：

- x++ 先计算x的值再将它加1。返回给外部表达式的值是加1前的原始值。
- ++x 先将x的值加1再使用新值作为整个++操作所得的值。

--运算符也类似，但为减1而非加1。

472
~
473

假如执行以下程序：

```
main()
{
    int x, y;

    x = 5;
    y = ++x;
    printf("x = %d, y = %d\n", x, y);
}
```

结果为

```
x = 6, y = 6
```

如果程序写为

```
main()
{
    int x, y;

    x = 5;
    y = x++;
    printf("x = %d, y = %d\n", x, y);
}
```

则最后结果为

```
x = 6, y = 5
```

语言中最常用的一种结构，即表达式：

```
*p++
```

要确定该表达式意义，首先需要解决的问题就是操作的顺序是什么。*运算符和++运算符会争夺操作数p。按照语言的优先级规则和结合性规则，该表达式可能会等同于 475

```
(*p)++
```

或者是

```
*(p++)
```

当这种情况发生时，C语言中的一元运算符按照从右至左的顺序进行运算。所以，++运算优先于*运算，第二种解释是正确的。但是你还是需要确定表达式到底是什么意思。

如以前所见，后缀式++运算符让p值先自增，随后再返回自增操作前的p值。由于p是一个指针，你必须根据指针运算定义自增操作，所以，在计算p+1时，你知道结果值应该指向数组中的下一个元素。如果p原来指向arr[0]，那么自增操作就会使其指向arr[1]。于是表达式

```
*p++
```

表示下面的含义：

> 间接引用指针p，使其当前指向的对象作为左值返回。其负面影响是，p值进行了自增，所以如果原左值是数组的一个元素的话，新的p值指向数组中的下一个元素。

但是为了理解该运算符的有用之处，必须更细致地考虑指针和数组之间的关系。

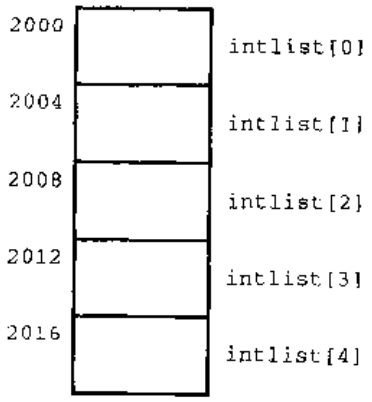
13.4.4 指针和数组的关系

C语言最不寻常的特征中，最有趣的是数组名作为指向该数组第一个元素的指针的同义词。这个概念很容易用例子解释。

声明

```
int intlist[5];
```

为一个含有5个整型数的数组分配了空间，如右图所示，分配的空间在计算机内存的某处。图中左边的地址是任意的。在这个图中，起始地址是2000，但也可以是其他的任意地址。



intlist代表一个数组，但也可以直接用作指针值。当它作为指针使用时，intlist定义为数组中第一个元素的地址。对于任意数组arr来说，下述等式在C语言中是永远成立的：

```
arr 等同于 &arr[0]
```

给定任何数组名，可以将它的地址直接赋给任何指针变量。当一个数组从一个函数传递到另一个函数时通常可以用到这个等式。典型地，被调用函数利用第12章中Sort原型所示的句法来声明这个数组：

```
void SortIntegerArray(int array[], int n);
```

如果你想要用上面定义的intlist数组调用Sort，那么传递给Sort的形式参数array的值将会是intlist中第一个元素的地址。如果将原型写成

```
void SortIntegerArray(int *array, int n);
```

节的内存，可调用

```
malloc(10)
```

结果会返回一个指针，指向一个10字节大小的内存块。为了使用新分配的空间，必须将malloc的结果存放在一个指针变量内，以后就可以像使用数组一样使用该指针变量了。

13.5.1 Void*类型

但在给出应用malloc函数的例子之前，先让我们讨论一个很可能会十分棘手的问题。在C语言中，指针是具有类型的。如果通过

```
int *ip;
```

478

声明一个变量ip，该变量在概念上是一个指向整型的指针。而声明

```
char *cp;
```

则引入了一个指向字符的指针变量cp。在ANSI C中，这些类型是不同的，如果你想要把其他类型赋给这些指针变量的话，编译器会发出警告信息。

指针有类型的概念引发了一个很有趣的问题，即malloc函数返回的是什么类型的指针呢？malloc函数用于为调用函数需要的任何类型的值分配新的内存空间，所以必须返回一个未确定类型的“通用”指针。在C语言中，通用指针类型是一个指向空类型void的指针，void也用于指示无返回值或参数列表为空的函数。如果声明一个指向void类型的指针，如

```
void *vp;
```

你可以将任何类型的指针值存入该变量，但不允许用*运算符间接引用vp。编译器不知道vp的基本类型是什么，所以没有办法谈论vp指向的值。

虽然如此，void*类型仍是有用的，因为它允许函数，特别是malloc函数，返回以后由调用函数创建实际类型的通用指针。malloc函数返回类型为void*的指针值，表明其原型为[⊖]：

```
void *malloc(int nBytes);
```

注意，这个函数原型的结果类型与指针变量的声明方法非常相似。*表示结果是和函数名而非基本类型相联系的指针，尽管在概念上，结果是与基本类型而非函数名联系的。除非习惯C语言中返回指针值的函数句法，否则是很容易混淆的。

ANSI C能在指向void的指针类型和指向基本类型的指针类型间自动进行转换。例如，如果声明字符指针cp为：

```
char *cp;
```

就可以用语句

```
cp = malloc(10);
```

将malloc的结果直接赋给cp。许多程序员在赋值之前，会使用下列强制类型转换将malloc返回的结果转换成指向字符的指针。这样做的一部分原因是因为历史因素，另一部分原因是这样做能使指针类型间的转换更清楚。

479

```
cp = (char*)malloc(10);
```

不管有没有明确使用强制类型转换，这条语句均可分配10个字节的新内存空间，并将第一个

⊖ malloc的参数类型实际上是在头文件stddef.h中定义的size_t类型。在一些机器上，int类型的大小可能不足以表示内存块的大小。不论在哪种情况下，参数在概念上是整数。

因为动态分配往往会频繁用于多种程序，因此错误检查尽管重要，却也是极其单调乏味的。而且常常因为出错消息使代码变得混乱不堪，搞乱算法结构。实际上，当空间用完时，就无计可施了，程序显示出错消息并且停止运行通常是唯一可行的方法。所以，调用malloc并将其嵌入一个包括内存不够测试的新抽象层中是很有用的。

genlib.h接口输出函数GetBlock，该函数结合了malloc的作用，并对NULL结果进行测试。如果GetBlock函数检测到内存不够的情况，就会调用Error函数。所以，如果以上两行代码用

```
arr - GetBlock(10 * sizeof(int));
```

代替的话，GetBlock函数的执行就结合了分配内存和检测错误的操作。因为GetBlock使内存分配的操作更清晰，程序更易懂，因此本书多采用GetBlock而不是malloc。但在概念上，这两个函数的功能是相同的。

为了进一步简化分配动态数组的过程，genlib.h库还定义了函数NewArray。NewArray取元素数和其基本类型后返回一个指针，指向特定大小的动态数组。所以，要分配一个含50个字符串的动态数组，应调用NewArray(50, string)。

13.5.4 释放内存

保证不会发生内存不够的一种方法是一旦使用完已分配的空间就立刻释放它。标准ANSI库提供了函数free，用于归还以前由malloc分配出去的堆内存。例如，如果你能肯定已经不再使用分配给arr的内存，就可以通过调用

```
free(arr);
```

481

释放该空间。但由于本书应用GetBlock执行动态分配，所以为分配和释放建立一个包括兼容的命名机制的统一抽象体系尤为重要。为此，genlib.h接口也包括一个FreeBlock函数，其操作方法和free完全相同。

但事实证明，知道何时释放一块内存并不那么容易。根本问题在于分配和释放内存的操作分别属于接口两边的实现及客户。实现知道何时该分配内存，返回指针给客户，但它并不知道何时客户结束使用已分配的对象，所以释放内存是客户的责任，虽然客户可能并不十分了解对象的结构。

对现在的大多数计算机的内存来说，你可以随意分配所需内存而不需要考虑释放内存的问题。这种策略几乎对所有运行时间不长的程序还是有效的。内存有限的问题只有在设计一个需要运行很长时间的应用，比如所有其他系统所依靠的操作系统时，才变得有意义。在这些应用中，不需要内存时释放它们是很重要的。

某些语言支持那些能主动检查正在使用的内存，并释放不再使用的内存的动态分配系统。此策略称为**碎片收集**（garbage collection）。C语言中也有碎片收集分配器，而且它未来的应用也许会更广泛。如果是这样的话，即使在长时间运行的程序中，也可以忽略释放内存的问题，因为你可以依靠碎片收集器自动执行释放内存的操作。

在大部分情况下，本书假定无论何时都可以给你的应用分配内存，这样大大简化了程序，并能使你专注于算法细节。

小结

本章介绍了指针的概念。指针是计算机内存中一个数据值的地址。指针通常是很有用的，

```
double d1;
double d2;
double *dp1;
double *dp2;
```

在图中，跟踪这些语句的操作：

```
dp1 = &d1;
dp2 = &d2;
*dp1 = 3.14159;
d2 = 2.71828;
dp1 = dp2;
*dp1 -= d2;
```

7. 判断题：对于任意变量x，表达式*&x都为x的同义词。

8. 判断题：对于任意变量x，表达式&*x都为x的同义词。

9. 常量NULL的内部表示是什么？

10. 间接引用NULL指针时会发生什么？

11. 引用调用的含义是什么？

12. 在不了解函数ApplySpecialOperation的作用的前提下，确定整型变量x和y中哪一个可能受到调用

```
ApplySpecialOperation(&x,y)
```

的影响。

13. 在什么情况下会使用引用调用？

14. 假设intArray声明为

```
int intArray[10];
```

且j是一个整型变量，描述计算机确定下列表达式的值时将采取的步骤。

```
&intArray[j+3];
```

15. 如果arr被声明为一个数组，描述以下两个表达式之间的区别：

```
arr[2]
arr+2
```

484

16. 假设double类型的变量在你使用的计算机系统中占8个字节。如果数组doubleArray的基地址为1000，那么doubleArray+5的地址值是什么？

17. 假如整型变量x和y的值分别为2和5，试确定以下几种情况下表达式的值和执行表达式后对x和y的值的影响。（在回答这个问题时，每一表达式都单独处理，而不是将其作为连续的表达式。换句话说，每次计算前x和y的值都是2和5。）

a. x++

b. -x

c. x++ + ++y

d. y + = x--

18. 给出一个包含++运算符的表达式，不同机器解释这个表达式会产生不同的结果。

19. 判断题：如果p为指针变量，那么表达式p++在p的内部表示中增加了1。

20. 表达式

```
*p++
```

```
Enter the elements of the array, one per line.
Use -1 to signal the end of the list.
? 67↵
? 78↵
? 75↵
? 70↵
? 71↵
? 80↵
? 69↵
? 86↵
? 65↵
? 54↵
? 76↵
? 78↵
? 70↵
? 68↵
? 77↵
? -1↵
The range of values is 54-86
```

4. 重写图12-5表示的选择排序的实现，将所有的数组引用改成指针引用。完成后，方括号字符“[”和“]”不应该出现在源文件中。但即便如此，SortIntegerArray函数的作用应和以前所述相同。

5. 写一个函数IndexArray(n)，返回指向动态分配的含n个元素的整型数组的指针，每个元素均初始化为自身的下标。例如，假设ip声明成：

```
int *ip;
```

那么语句

```
ip = IndexArray(10);
```

应该产生如下内存配置：



6. 写一个函数Tabluate(array, n)，使其取一个整型数组和它的有效大小作为参数，画出一张显示每个元素出现次数的表格。例如，假设输入数组为

91	93	98	92	92	95	93	92	91	95	99	92	98

则程序应能产生如下表格：

```
91: 2
92: 4
93: 2
95: 2
98: 2
99: 1
```

第14章 再论字符串

. . . Untune that string
And, hark! what discord follows;

— Shakespeare, *Troilus and Cressida*

学习目标

- 理解字符串数据的基本表示方法。
- 了解字符串变量声明为数组和声明为指针之间的区别。
- 能够使用ANSI字符串库中的函数。
- 学会如何用string.h函数实现strlib.h接口。

490 ~ 491

从第2章我们就开始使用字符串了，从第9章开始已经能够执行高级的字符串操作。但是到目前为止，本书中的程序只将string作为一种抽象的类型。作为一种抽象的类型，string类型的值是一个完整的、不可分割的实体。可以将字符串传递给函数，也可以将字符串作为结果返回，还可以将一个字符串的值赋给另一个字符串。如果想要从字符串中选出一个字符或是进行更加复杂的字符串操作，就必须调用strlib.h接口导出的某个函数。

和其他接口一样，strlib.h的主要目的之一是对客户隐藏那些不必知道的细节。但是，由于字符串操作是许多程序设计应用的重要方面，因此必须学会在较低的细节层次上使用字符串。为了完全弄清字符串的工作原理，有必要超越抽象的层次，思考其根本的表示方法。

14.1 string类型的概念表示

在C语言中，为了更有效地使用字符串，必须能从三个角度考虑它：

1) 作为字符数组。

2) 作为指向单个字符的指针。

3) 作为具有整体的概念完整性的完整实体。

这三个观点不是矛盾的，而是互补的，每一个观点都描述了一个现实的角度。对于一个存储在计算机中的字符串，可以选择这三种观点中的任何一种来考虑。因为只是从不同的角度来考虑字符串，因此字符串本身并不发生变化，唯一改变的只是在你的概念中如何表示字符串。

14.1.1 字符串作为数组

在计算机内部，字符串被表示为字符数组。只要将一个字符串存储到内存，这个字符串中的字符就都被分配到连续的字节中。例如，当编译器在一个程序中看到字符串常量"Hello"时，就会把组成这个字符串的字符存入内存中连续的字节空间内。

但是，存放字符本身并不足以表示所有与字符串有关的重要信息。操作字符串值的程序必须具有某种确定字符串结束的方法。如果编译器要将字符串常量"there"中的字符直接存储在字符串常量"Hello"的字符之后，内存看起来如下图所示：

492

```
    ...操作str[i]的循环体...
}
```

在每个循环周期中，选择表达式str[i]指的都是字符串中的第i个字符。由于循环的目的是处理每一个字符，循环会一直持续到str[i]选择了标志字符串结束的空字符。所以这个for循环在功能方面等价于第9章中介绍的如下习语：

```
for (i = 0; i < StringLength(str); i++) {
    ...操作IthChar (str, i) 的循环体...
}
```

利用数组选择形式比使用strlib.h接口中的函数有效得多。首先，数组选择避免了调用IthChar函数所需付出的代价。而且，用

```
i < StringLength(str)
```

作为for循环控制语句中的测试条件意味着，即使字符串长度通常是不变的，程序也必须在每个循环周期中都重新计算一遍字符串长度。通过检查结束标记，程序可以避免这些冗余的计算。 493 到 494

为了对这两种方法进行比较，可以不采用任何strlib.h接口中的函数来重写图10-5中的Pig Latin一例中介绍的FindFirstVowel函数。图14-1中是第10章中原来的实现。

如果以数组符号写这个函数的话，它的实现如图14-2所示。

这两个程序在算法上是完全相同的，只在代码的细节上有些微区别，主要是检测字符串结束和选择每个字符的方法有所不同。

```
int FindFirstVowel(string word)
{
    int i;

    for (i = 0; i < StringLength(word); i++) {
        if (IsVowel(IthChar(word, i))) return (i);
    }
    return (-1);
}
```

图14-1 用抽象字符串实现FindFirstVowel

```
int FindFirstVowel(char word[])
{
    int i;

    for (i = 0; word[i] != '\0'; i++) {
        if (IsVowel(word[i])) return (i);
    }
    return (-1);
}
```

图14-2 基于数组的FindFirstVowel的实现

14.1.2 字符串作为指针

和C语言中的任何数组一样，字符数组也可以被解释为指向其首元素的指针。给定一个指向字符串值的指针，就可以用指针运算引用该字符串中的各个字符。因此，FindFirstVowel也可以写成如图14-3所示的代码。

符。由于该函数的实现使用数组符号，所以它的形参被声明为如下的字符数组：

```
int FindFirstVowel(char word[])
```

图14-3中的实现将参数作为指针，如函数头一行的声明所示：

```
int FindFirstVowel(char *word)
```

就C语言编译器而言，这三个函数头中的参数声明是完全一样的，可以互换使用。选择某个表示方法的价值在于这种表示方法传达给读者的额外信息。通常，声明形参应该和这些参数的使用风格一致。如果把字符串作为一个字符数组考虑，并用方括号选择各个字符的话，应该将该字符串声明成一个数组。如果把字符串作为一个指针考虑，并用*间接引用这个指针的话，应该将该字符串声明成一个指针。如果把字符串作为一个完整实体考虑，并且这个完整实体比其各个组成部分更重要的话，最好的方法是将该字符串声明成string类型。

14.1.5 字符串变量

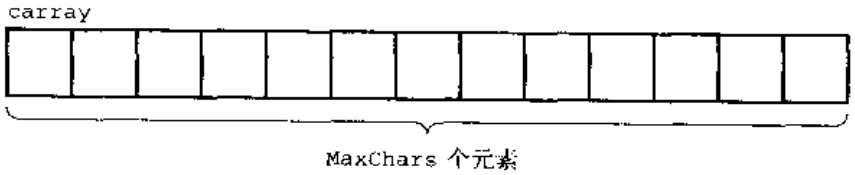
尽管在C语言中数组和指针经常可以互换使用，我们必须认识到除非变量是上一节描述的形参，否则将这个变量声明成数组和将其声明成指针是截然不同的。对于所有其他的变量类型来说，所选择的声明方式决定了内存的分配方式。

如果用以下代码来声明一个数组：

```
char carray[MaxChars];
```

497

那么为carray分配的空间就在当前函数帧内。如下图所示，你有MaxChars个字节可用于存放字符：



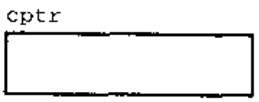
由于内存是显式分配的，因此可以直接给carray的元素赋值，也可以将carray数组传递给其他函数，这样就能自由操作分配到的内存了。

如果你换一种方式，将一个变量cptr声明为

```
char *cptr;
```

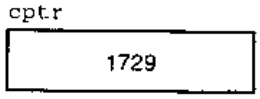
或者（等同地）声明为

```
string cptr;
```



则完全没有为字符分配任何内存。变量cptr只是一个指针变量（如右上图所示）。

因为这一变量没有显式地初始化，所以它的内容是不确定的。它可能包含0、42或任何以前此位置被使用时所留下的随机值。假设cptr中包含的正好是1729，如右下图所示。如果改变cptr[0]的值，将会改变地址1729里所存储的值。但是，地址1729可能被程序中的某个变量或某一部分代码使用。这样进行赋值的话会很容易破坏关键数据并使程序崩溃。



常见错误 声明一个指针变量时，必须确定你已对其进行了初始化，这样才能让它指向内存的真实位置。未成功初始化指针是造成错误的常见原因。这些错误是很难发现的，因为编译器不能捕捉到它们。最好的方法是完全地避免它们。

使用cptr之前，你必须提供必要的字符内存，方法是给其赋一个和可用内存区域的地址

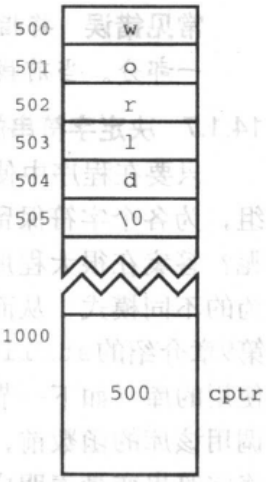
来分配字符串内存的话，形如

```
carray = "hello";          (这条语句是非法的)
```

的赋值语句是非法的。因为变量carray不是左值，你无法给它赋值。

将字符串声明成指针就没有这种限制了。例如，声明char *cptr;后，cptr="world";是完全合法的。

常见错误 在C语言中，数组名不是左值。所以把值赋给任何数组名都是非法的，你必须将值赋给单个数组元素。特别是，如果你把一个字符串声明为字符数组，就不能直接将字符串值赋给该数组。



编译器将"world"的字符存在内存某处，赋值语句将字符串第一个字符的地址存储在cptr中。例如，如果编译器将"world"的字符存放在从500开始的内存位置，给cptr赋值的效果可以用右图表示。变量cptr只是经过更新指向了一个新的位置。

调用函数时也有类似的限制。尽管函数可以将一个数组作为参数，但在C语言中函数返回一个数组是非法的。然而，函数可以返回指针，从而达到同样的效果。如果曾经在函数里用动态分配为某个数组分配内存，则可以将指向该内存的指针作为函数的值返回，将结果赋给一个指针变量，接着就能像使用数组一样使用这个指针变量。

举例来说，strlib库中的函数CharToString可以用如下方式实现：

```
string CharToString(char ch)
{
   char *cptr;

   cptr = GetBlock(2);
   cptr[0] = ch;
   cptr[1] = '\0';
   return (cptr);
}
```

第一条语句从堆中分配了两个字节的内存：一个字节给字符，另一个字节给每个字符串末尾都必须出现的空字符。接下来两条语句把正确的值赋给这两个字符。最后一条语句将一个指向新分配内存的指针作为函数值返回。接着，客户写形如

```
star = CharToString('*');
```

的语句就是合法的，该语句假设star被声明为字符指针，或者（等同地）被声明成抽象类型string。

认识到这个方法只适用于动态分配数组空间是很重要的。如果想要返回一个指向在函数中声明的数组的指针，程序将会失败。问题在于分配给声明成局部变量的数组的内存在函数返回时就自动释放了。

要理解这个问题，想象你用以下这个错误的方法实现CharToString：

```
string CharToString(char ch)
{
```

任务，但学会怎样使用string.h接口不仅仅是记住一组新的函数名。string.h接口是建立在一个不同的字符串操作的模型上，要求你从一个新的角度看待字符串。

两个字符串库的最大区别在于怎样为字符串中的字符分配内存空间。在strlib.h接口中，函数本身就会分配必要的内存空间。作为客户，你无需为分配的细节担忧，这样就使得strlib.h接口更容易使用。string.h接口中的函数将分配内存的重任转移到了客户身上。当一个新字符串需要内存时，客户要负责提供内存。为此，客户通常会声明一个足以保存结果的字符数组，并将它作为参数传递给库函数。然后该实现会把新字符串值写入调用函数提供的内存空间中。

表14-1 string.h输出的常用函数

函数调用	操作
strcpy(dst, src)	将字符从src复制到dst
strncpy(dst, src, n)	最多将n个字符从src复制到dst
strcat(dst, src)	将src追加字符至dst后面
strncat(dst, src, n)	最多将n个字符从src追加至dst后面
strlen(s)	返回s的长度
strcmp(s1, s2)	返回一个整数，表示比较的结果
strncmp(s1, s2, n)	类似于strcmp，但最多比较n个字符
strchr(s, ch)	返回一个指针，指向s中的第一个ch的实例（或者返回NULL）
strrchr(s, ch)	返回一个指针，指向s中的最后一个ch的实例（或者返回NULL）
strstr(s1, s2)	返回一个指针，指向s1中的第一个s2的实例（或者返回NULL）

502 ~ 503

14.2.1 strcpy函数

string.h接口中的strcpy函数为库中函数如何给调用函数返回一个新的字符串值提供了最好的方法。strcpy函数将一个源（source）字符串中的字符复制到另一个目标（destination）字符串中。为了和赋值运算符相一致，复制操作是从右向左进行的，strcpy将目标参数作为第一个参数，因此就有了以下原型[⊖]：

```
void strcpy(string dst, string src);
```

如果你想要操作一个字符串，同时又保留它的原值的话，strcpy函数是很有用的。例如，假设字符串变量line含有一行从用户处读入的文本，程序要求你对line中的字符做一些修改。直接用数组选择来更改line中的元素会破坏line原来的内容，而且你以后还有可能会用到这些内容。所以，为了避免改变line，可以将其中的各个字符复制到另外一个数组再进行操作。

保存数据中间副本的数组称为**缓冲区**（buffer）。在这种情况下，我们需要的是一个字符缓冲区，以便装入line的数据副本。当使用ANSI字符串库时，通常的方法是明确声明一个字符数组作为缓冲区，大小足以存放应用中可能遇到的最大字符串。所以，如果常量MaxLine含有最长输入串的长度，你可以这样声明缓冲区：

```
char buffer[MaxLine+1];
```

此处的+1用来为长度正好是MaxLine的字符串末的空字符留出空间。

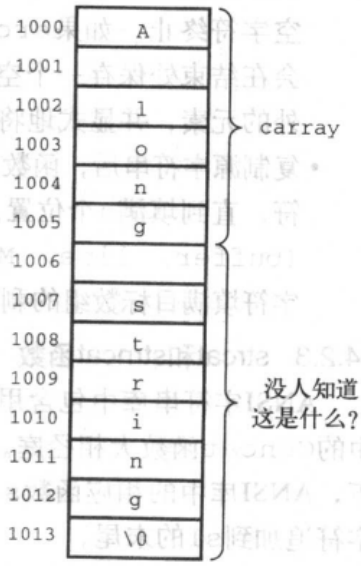
重要的是，要牢记不能采用如下赋值语句将line的字符复制到buffer中：

⊖ 和库中的定义一样，strcpy函数确实返回了目标数组的地址，尽管大多数客户会忽略这个值。为了表达的简洁性，本文将strcpy函数定义成一个无结果的过程。

```
char carray[6];

strcpy(carray, "A long string");
```

执行这一程序时会将"A long string"的前六个字符正确复制到carray中，但是，程序 505
又继续把剩余的字符复制到内存中carray之后的字节里面，所以，调用strcpy之后，内存如右图所示。地址1006～1013间的字节存放的也许是其他变量或是程序本身的一部分，所以改变它们肯定会引起程序失败。写入的数据超出作为缓冲区的数组的大小是一种常见的程序设计错误，这种错误称为**缓冲区溢出**（buffer overflow）。



如果目标字符串是一个未妥善初始化的指针变量，就会发生同样严重的问题。例如，假设用语句

```
string str;
```

声明了一个字符串变量str，那么就不能用strcpy复制字符串到该变量中。如果试图写语句

```
strcpy(str, "A long string");        (此语句无效)
```

并且没有事先初始化变量str使其指向某一字符数组，"A long string"会被复制到内存中无法预知的区域。

作为程序员，你的责任是确保不会发生这些错误。如果你认为已经分配给目标字符串的空间可能不够，就必须在使用strcpy复制字符串之前检查数据的长度。所以，在用strcpy将line的内容复制到buffer之前，应该通过测试语句行确保不会发生缓冲区溢出的情况，测试如下：

```
if (strlen(line) > MaxLine) Error("Input line too long");
strcpy(buffer, line);
```

506

如果没有检测到缓冲区溢出的话会发生可怕的后果。1988年，康奈尔大学的一名研究生发布了一个程序，它像病毒一样，通过连接在因特网上的计算机间迅速传播。这个程序就是我们所知道的因特网蠕虫病毒。因特网上许多电脑的系统设备程序都不会检查机器内部缓冲区的字符内存是否被输入数据用尽，蠕虫病毒正是利用了这一点，通过输入大量数据用尽已分配内存，然后覆盖系统程序本身，并执行利于自身的命令。

14.2.2 strncpy函数

为了避免在使用strcpy函数时缓冲区溢出，ANSI字符串库包含了另一种形式的strcpy，称为strncpy。strncpy函数允许客户指定一个长度限制，其原型如下：

```
void strncpy(string dst, string src, int n);
```

和strcpy一样，strncpy将src指定的字符串中的字符复制到dst指定的字符数组中。区别在于strncpy最多从src复制n个字符，遇到空字符会提前停止。strncpy允许你指定字符串的最大长度，所以能防止写入的字符超过字符数组的大小。例如，如果通过语句

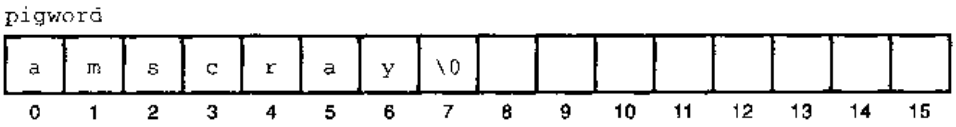
```
char buffer[MaxLine+1];
```

声明字符数组buffer，则可以用函数调用

最后一步，调用 508

```
strcat(pigword, "ay");
```

完成Pig Latin词汇，结果如下：



和使用strcpy一样，你必须小心使用strcat函数，以避免连接过多字符，从而超出缓冲区所能提供的大小。字符串库包含的函数strncat有一定的作用，但是决不如你想象的那么多。调用strncat(s1, s2, n)最多能从s2复制n个字符到s1末尾，但为了确定给n赋予何值，还需检查s1的当前长度，这样才能知道缓冲区还剩余多少空间。

14.2.4 strlen、strcmp和strncmp函数

string.h输出的strlen和strcmp函数与第9章介绍的StringLength和StringCompare完全一样。使用ANSI库时，可以像使用StringLength和StringCompare一样使用strlen和strcmp。

但string.h接口没有包含与StringEqual直接对应的函数。要测试两个字符串是否相等，C语言程序员只需调用strcmp并查看结果是否为0。当程序能明确地将结果和0作比较时，这个方法不会引起混淆。但如果程序员依赖C语言中把整数0解释成FALSE，而把其他整数解释成TRUE的话，程序就难以读懂了。例如，有时会看到如下if语句：

```
if (strcmp(s1, s2)) {
    ...语句...
}
```

当s1和s2不相同时，就会执行if语句的内容。

图12-2中给出的FindStringInArray的实现给出了紧凑编程风格的含义更为模糊的例子：

```
int FindStringInArray(string key, string array[], int n)
{
    int i;

    for (i = 0; i < n && strcmp(key, array[i]); i++);
    if (i < n) return (i);
    return (-1);
}
```

只要i比n小，且key与当前数组项不符，for循环就会一直执行下去。在两种情况下可以退出for循环：数组中的元素都已测试完毕，或者是循环在下标i处找到了匹配的元素而终止。由于for控制行已包括了所有所需的工作，所以循环体只用了一个分号。 509

string.h接口也包含另一个字符串比较函数strncmp，它有时会非常有用。调用strncmp(s1, s2, n)和调用strcmp(s1, s2)相似，只是strncmp最多只考虑两个字符串中的n个字符。如果字符串的前n个字符都相同，strncmp就返回0，就算以后的字符不同也是如此。

14.2.5 strchr、strrchr和strstr函数

string.h接口也输出一些用于搜寻字符串的函数。函数strchr(s, ch)和FindChar(ch, s, 0)的功能相同。它们主要的区别是，strchr返回一个指向匹配字符的指针，而FindChar返回该字符的下标。如果没有找到匹配字符，strchr返回NULL。函数strrchr

以便开始学习标准接口。

在这种情况下，通常的做法是让一个接口重新命名比它更低层的接口的函数。这样的函数就称为**转换函数**。提供一个传递函数而不让客户使用较低层次的对应函数的目的是为了减少客户在概念上的复杂性。如果你不得不使用一些来自strlib.h和string.h的函数，就必须将两个库接口同时包含在程序中。更重要的是，你不得不同时理解这两个接口和它们所提供的函数。定义了strlib.h，你就可以在准备使用较低层的接口之前不需要学习低层接口。 511

```
/*
 * File: invert.c
 * --------------
 * This file implements a function InvertName(result, name)
 * that takes a name in standard order (first middle last) and
 * returns a new string in inverted order (last, first middle),
 * which makes it easier to alphabetize the names. The test
 * program reads in names and prints out their inverted form,
 * stopping when a blank line is entered.
 */

#include <stdio.h>
#include <string.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constant
 * --------
 * MaxName -- Maximum number of characters in a name
 */

#define MaxName 40

/* Private function prototypes */

static void InvertName(char result[], char name[]);

/* Main program */

main()
{
    char *standardName;
    char invertedName[MaxName+1];

    printf("This program converts a name in standard order\n");
    printf("into inverted order with the last name first.\n");
    printf("Indicate the end of input with a blank line.\n");
    while (TRUE) {
        printf("Name: ");
        standardName = GetLine();
        if (strlen(standardName) == 0) break;
        InvertName(invertedName, standardName);
        printf("%s\n", invertedName);
    }
}
```

图14-4 invert.c

```
int StringCompare(string s1, string s2)
{
    return (strcmp(s1, s2));
}
```

14.3.2 strlib分配函数的实现

strlib.h接口里大多数函数的主要特征是它们动态地为新创建的字符串分配内存。为了集中这一阶段的操作，strlib.c程序定义了一个专用函数CreateString(len)，用于给长度为len的字符串分配内存，而且还考虑到在字符串最后为'\0'预留一个额外的字节。和大多数软件包一样，CreateString的实现可以写为：

```
static string CreateString(int len)
{
    return ((string) GetBlock(len + 1));
}
```

一旦有了这个函数，你可以用它来写strlib中所有需要分配内存的函数。例如，考虑如下CharToString的实现，它为一个只含一个字符的字符串分配内存：

```
string CharToString(char ch)
{
    string result;

    result = CreateString(1);
    result[0] = ch;
    result[1] = '\0';
    return (result);
}
```

函数动态地分配空间来存放只有一个字符的字符串，然后通过赋入特定的字符和终止空字符来初始化该内存。

512 ~ 514

让我们看一个更实质性的函数，如Concat。一种方法是显式地编写所有关于复制的代码。该实现先为连接后的字符串分配空间，再依次从每个字符串复制字符，如下所示：

```
string Concat(string s1, string s2)
{
    string s;
    int len1, len2, i;

    len1 = strlen(s1);
    len2 = strlen(s2);
    s = CreateString(len1 + len2);
    for (i = 0; i < len1; i++) s[i] = s1[i];
    for (i = 0; i < len2; i++) s[i + len1] = s2[i];
    s[len1 + len2] = '\0';
    return (s);
}
```

尽管这个实现能工作，但它并没有利用较低层库中的函数。通常，ANSI字符串库中函数的设计应使得它们尽可能利用所运行的计算机的硬件特征而有效地工作。如果在你的实现中使用了这些函数，那么产生的代码常常会比采用显式字符串操作的程序更有效。所以，将strcpy作为实现的一部分使用是很有意义的，程序如下：

```
string Concat(string s1, string s2)
{
```

复习题

1. 本章要求你从三个不同的角度考虑字符串。这三个角度是什么？

2. 根据基本表示方法，怎样确定一个字符串的末尾？

3. 若将字符串看成字符数组的话，你会使用什么习语来处理每一个字符？如果将字符串看成指向字符的指针的话，习语又会发生什么改变？

4. 判断题：将参数声明为字符数组和将其声明为指针是等效的。

5. 判断题：将局部变量声明为字符数组和将其声明为指针是等效的。

6. C语言对数组变量的使用有何限制？

7. 为什么将指向局部数组的指针作为函数返回值是错误的？

8. string.h和strlib.h接口所表示的抽象模型有哪些主要区别？

9. 如果调用strcpy(s1, s2)，哪个参数是源字符串，哪个参数是目标字符串？

10. 作为客户，在调用strcpy之前应该先做什么？

11. *缓冲区溢出*是什么意思？

12. 限制函数strncpy使用的两个因素是什么？

13. 使用ANSI字符串库时，怎样达到使用StringEqual函数的效果？

14. strlib.h中的哪两个函数是作为转换函数实现的？将这两个函数包含在接口中有什么重要意义？

15. 为什么在实现Concat时，使用函数strcpy比用for循环语句更好？

程序设计练习

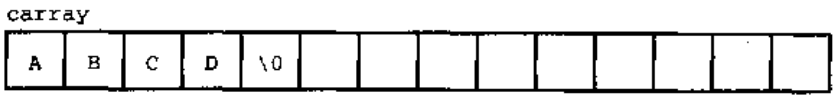
1. 字符串库中最简单的函数是strlen，它通过寻找标志字符串结束的空字符来计算字符串的长度。编写一个strlen的实现，它以一个字符数组为参数。 517

2. 重写练习1的实现，使strlen的实现以一个指向字符的指针作为参数，而且只采用指针操作来查看字符串中的每一个字符。

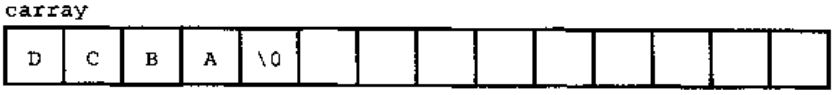
3. 写一个strcmp函数的实现，使其不调用其他函数，直接使用参数的基本表示。作为附加练习，以两种方式实现该函数，一种方式是只使用数组操作，另一种方式是只使用指针操作。

4. 第9章中的9.4.7节定义了函数Acronym(s)，该函数通过组合字符串s中每个单词的首字母来返回一个新的词，假定字符串s中包含的单词都用一个空格分隔。函数Acronym的设计是符合strlib.h模型的，因为它分配并返回了一个新的字符串。为了满足string.h接口的规则，这样的函数必须通过将数据写入调用函数提供的字符串缓冲区来传递结果。设计并实现一个采用此策略来计算词首字母的函数。你的函数不应包含strlib.h或者使用它的函数。

5. 写一个函数ReverseSting(carray)，使参数中的字符倒序排列。例如，如果carray原来包含的数据为：



调用ReverseString(carray)后，应该将字符倒序排列，使数组内容变为：



```
The word now looks like this: ----
You have 4 guesses left.
Your guess: U↵
That guess is correct.
The word now looks like this: -U--
You have 4 guesses left.
Your guess: S↵
There are no S's in the word.
The word now looks like this: -U--
You have 3 guesses left.
Your guess: N↵
That guess is correct.
The word now looks like this: -UN-
You have 3 guesses left.
Your guess: K↵
That guess is correct.
The word now looks like this: -UNK
You have 3 guesses left.
Your guess: H↵
There are no H's in the word.
The word now looks like this: -UNK
You have 2 guesses left.
Your guess: B↵
There are no B's in the word.
The word now looks like this: -UNK
You have only one guess left.
Your guess: J↵
That guess is correct.
You guessed the word: JUNK
You win.
```

图 14-5 (续)

520
~
521

```
What's in a name?
That which we call a rose
By any other name would smell as sweet.
```

文件可被视作一种二维结构，即由单个字符组成的行序列。例如，假设在你的语录集合中加入了一个新文件witches.txt，它包含出自《麦克白》的以下两行语句： 524

```
Double, double toil and trouble:
Fire burn and cauldron bubble.
```

大家很容易就认为witches.txt是由两行组成的。但从内部来说，文本文件是由一维的字符序列表示的，除了可见的打印字符外，文件还包含从第2章就开始使用的行终止符'\n'。所以，更准确地来说，文件witches.txt是以如下形式出现的：

```
Double, double toil and trouble:\nFire burn and cauldron bubble.\n
```

文本文件和字符串在很多方面有相似之处。文本文件和字符串一样，是由有序的字符集合组成的。而且，它们都有确定的结束点。在字符串中，数据结束是用空字符表示的；文本文件的结尾用一种特殊的文件结束标志，说明该标志之后再没有字符出现了。例如，witches.txt中bubble一词后的字符'\n'表示文件结束。当从某一文件中读取字符时，必须检测这个文件结束标志，从而来确定何时没有字符可读。

另一方面，字符串和文件有一些重要的区别。其中，最重要的区别就是数据能否被永久保存。字符串仅在程序运行的一段时间内临时存储在计算机内存中，而文件能被长期保存在存储设备中，直到被明确地删除或覆盖为止。使用字符串和文件中的数据的方式也是不同的。字符串是一个字符数组。指明适当的下标后，就可以以任何次序选择字符。在程序中，文件通常是顺序读写的。当程序读取一个已有文件时，会从头到尾读取字符。当程序创建新文件时，会从第一个字符开始，按照字符顺序逐个写入字符。

15.2 C语言中文件的使用

在ANSI C中，基本的文件操作是标准I/O库接口stdio.h的一部分。从第2章开始，已使用了标准I/O库，但相对而言很少有机会研究包含于其中的各种功能。标准I/O库的真正强大之处在于它允许你以可移植的方式指定文件操作，这种方式是方便、有效的。 525

在C程序中使用文件，需要完成以下工作：

1) 声明一个FILE*类型的变量。

2) 通过调用fopen函数将此变量和某实际文件相联系。这一操作称为**打开**（open）文件。打开一个文件要求指定文件名，并且指明该文件是用于输入还是输出。

3) 调用stdio.h中适当的函数完成必要的I/O操作。对输入文件来说，这些函数从文件中将数据读取至程序中；对输出文件来说，函数将程序中的数据转移到文件中去。

4) 通过调用fclose函数表明文件操作结束，这一操作称为**关闭**（close）文件，它断开了FILE*变量与实际文件间的联系。

接下来的四节将依次讨论以上的每个步骤。

```
while (TRUE) {
    printf("Input file name: ");
    filename = GetLine();
    infile = fopen(filename, "r");
    if (infile != NULL) break;
    printf("Can't open the file %s. Try again.\n", filename);
}
```

15.2.3 执行I/O操作

一旦打开一个文件，下一步要做的就是读出或写入实际数据。要完成这项工作，应该根据应用选择一种策略。最简单的方法就是使用getc和putc函数，逐个字符地读写文件。但 527
很多情况下，逐行处理文件更加方便。为了达到此目的，stdio.h接口提供了fgets和fputs这两个函数，但使用simpio.h所提供的ReadLine函数会更简单，该函数能解决使用ANSI标准函数时产生的很多问题。在更高层次上，可以选择使用fscanf和fprintf读写格式化数据，这样就可以将数字数据和字符串以及其他数据类型混合起来。上面提到的函数以后在本章还会再提到。之所以这样组织讨论是为了把基本规则相同的函数放在一起考虑。

15.2.4 关闭文件

不管选择哪种I/O操作策略，都必须确保关闭所有打开的文件。关闭文件通过调用fclose函数并配合适当的文件指针来完成。所以，关闭和变量infile、outfile相关的文件可用以下代码：

```
fclose(infile);
fclose(outfile);
```

尽管退出程序时会自动关闭所有打开的文件，但养成显式地关闭文件的习惯还是很有好处的，因为

- 程序的读者很容易知道什么时候文件在使用，什么时候不再需要使用了。
- 更容易将程序组合到自己开关文件的较大程序中。

15.2.5 标准文件

标准I/O库定义了三个特殊的标识符：stdin、stdout和stderr，它们作为FILE*常量可被所有程序使用，称为**标准文件**（standard file）。常量stdin指定标准输入文件，它是用户的输入源。常量stdout指出标准输出，表示写给用户的输出数据所在的设备。常量stderr代表标准错误文件，用于报告任何用户应该看到的出错消息。通常，标准文件都指向计算机控制台。当从stdin读取数据时，输入来源于键盘；当向stdout或stderr写入数据时，输出出现在屏幕上。但有些系统为了方便起见可以改变这些联系，所以标准输入会来源于某一文件，或者标准输出被分配给一些其他的文件或设备。

即使你所使用的系统不具备改变标准文件分配的能力，标准文件的存在还是有益处的，因为这表明了任何在本章描述过的函数都可以直接用控制台操作。使用stdin、stdout和stderr这些名字，你可以直接读写字符、行或是格式数据。 528

15.3 字符I/O

文件处理的最简便的方法是逐字符地遍历该文件。stdio.c接口定义了getc(infile)函数，该函数从某一文件中读取下一个字符，并将其返回给调用函数。同时还定义了getchar()函数，该函数从标准输入文件中进行读取，因此与getc(stdin)作用相当。

```
 * Usage: CopyFile(infile, outfile);
 * ---------------------------------
 * This function copies the contents of infile to outfile. The
 * client is responsible for opening these files before calling
 * CopyFile and for closing them afterward.
 */

static void CopyFile(FILE *infile, FILE *outfile)
{
    int ch;

    while ((ch = getc(infile)) != EOF) {
        putc(ch, outfile);
    }
}

/*
 * Function: OpenUserFile
 * Usage: fileptr = OpenUserFile(prompt, mode);
 * --------------------------------------------
 * This function prompts the user for a file name using the
 * prompt string supplied by the user and then attempts to
 * open that file with the specified mode. If the file is
 * opened successfully, OpenUserFile returns the appropriate
 * file pointer. If the open operation fails, the user is
 * informed of the failure and given an opportunity to enter
 * another file name.
 */

static FILE *OpenUserFile(string prompt, string mode)
{
    string filename;
    FILE *result;

    while (TRUE) {
        printf("%s", prompt);
        filename = GetLine();
        result = fopen(filename, mode);
        if (result != NULL) break;
        printf("Can't open the file \"%s\"\n", filename);
    }
    return (result);
}
```

图 15-1 (续)

copyFile中的while循环非常常用，值得我们仔细思考。while循环的判断表达式通过嵌入式赋值语句将读入字符和检测文件结束标记的操作结合起来。当程序判断while循环条件时，首先从判断子表达式

```
ch=getc(infile)
```

开始，该表达式读入一个字符并将其赋给ch。在执行循环体之前，程序不能确保所赋的值不是EOF。赋值语句外的括号是必不可少的，否则该表达式将把character和EOF的比较结果赋值给ch，这是不正确的。这一循环的作用与如下所示的长表达式是相同的：

```
while (TRUE) {
```

此时，tempfile已经获得所需的最终内容。如果删除原有的名为witches.txt的文件，并 532
将tempfile重新命名为witches.txt，就得到了所需的唯一一个文件，它包含我们所希望得到的内容，如右图所示。在编写实现这一策略的代码时，需要使用三个定义在stdio.h头文件中的函数——tmpnam、remove和rename，这三个函数将在下文中进行描述。

witches.txt

```
DOUBLE, DOUBLE TOIL AND TROUBLE:
FIRE BURN AND CAULDRON BUBBLE.
```

虽然我们可以为临时文件取任何名字，但stdio.h接口中包括的函数tmpnam可以为临时文件自动生成文件名。文件命名的习惯因不同机器而异。调用函数tmpnam(NULL)会返回一个字符串，它的值适合作为该台机器上临时文件的文件名[⊖]。因此，可以按照如下代码创建并打开一个新的临时文件：

```
temp=tmpnam(NULL);
outfile=fopen(temp, "w");
```

删除一个文件只需调用函数remove(*name*)即可，此处的*name*是一文件名。重命名一个文件同样非常简单，可以通过调用函数rename(*old name*, *new name*)完成。和ANSI库中的很多其他函数一样，remove和rename在调用成功后返回0，调用失败后返回一个非零值。虽然有时会遇到将这些函数当作过程使用的情况，但检验返回值以确保调用成功是更安全的一种做法。

这三个函数提供了编写程序ucfile.c（如图15-2所示）所需的内容，该程序可将一个文件中的字符转换为大写字符。

```
/*
 * File: ucfile.c
 * --------------
 * This program updates the contents of a file by converting all
 * letters to upper case.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"

/* Private function prototypes */

static void UpperCaseCopy(FILE *infile, FILE *outfile);

/* Main program */

main()
{
    string filename, temp;
    FILE *infile, *outfile;

    printf("This program converts a file to upper case.\n");
```

图15-2 ucfile.c

⊖ 当函数tmpnam以NULL作为参数调用时，返回一个指向标准I/O库实现专用的内存的指针。因此在结束第一个临时文件之前生成第二个临时文件是不安全的。如果必须这样做，可以用CopyString或strcpy将tmpnam的结果复制到另外的内存中。

们全部都“推回”到输入文件。所幸的是，能够推回一个字符就已经可以满足绝大多数情况的需要。

为了理解函数ungetc是如何使用的，假设需要编写一个程序，将程序从一个文件复制到另一个文件，在此过程中删除所有的注释语句。众所周知，在C语言中，一条注释语句以字符序列/*开始，以序列*/结束的。删除注释语句的程序必须能在检测到开始标记/*前复制字符，在检测到后逐一读字符但不进行复制，直至检测到结束标记*/为止。

这个问题的难点在于注释标记是由两个字符组成的。如果每次从文件中复制一个字符，那么当遇到“/”时应该如何处理呢？它有可能是注释语句的开始，这时就不能将其复制到输出文件中。另外，它也有可能是除号。只有读入下一个字符后我们才能进行判断，如果下一个字符是“*”，那么要将两个字符都忽略掉，并标识出已经进入注释语句。如果不是*，则需将其推回到输入流中，留待下一个循环周期中再复制它。图15-3中所示的函数CopyRemoving-Comments实现了这一策略。

533 ~ 535

```
static void CopyRemovingComments(FILE *infile, FILE *outfile)
{
    int ch, nch;
    bool commentFlag;

    commentFlag = FALSE;
    while ((ch = getc(infile)) != EOF) {
        if (commentFlag) {
            if (ch == '*') {
                nch = getc(infile);
                if (nch == '/') {
                    commentFlag = FALSE;
                } else {
                    ungetc(nch, infile);
                }
            }
        } else {
            if (ch == '/') {
                nch = getc(infile);
                if (nch == '*') {
                    commentFlag = TRUE;
                } else {
                    ungetc(nch, infile);
                }
            }
            if (!commentFlag) putc(ch, outfile);
        }
    }
}
```

图15-3 CopyRemovingComments

15.4 面向行的I/O

由于文件通常被划分成行，因此很有必要一次读入整行数据。stdio.h中执行这一操作的函数为fgets，其函数原型为：

```
string fgets(char buffer[], int bufSize, FILE *infile);
```

此函数的作用在于将下一行文件读入字符数组buffer中。通常，fgets在读入第一个换行

同的内存，因此两者的值是相同的字符串。

常见错误 如果使用同一个字符缓冲区从同一个文件中读入多行，要记住在读入下一行数据之前，需要将数据从缓冲区中复制到其他存储空间。如果不进行这样的操作，前一行的内容将被覆盖。

解决这个问题的办法之一是利用strlib库中的函数CopyString将每一个字符串写入各自的堆内存中，如下所示：

```
void ReadTwoLines(FILE *infile)
{
    string line1, line2;
    char buffer[MaxLine];

    fgets(buffer, MaxLine, infile);
    line1 = CopyString(buffer);
    fgets(buffer, MaxLine, infile);
    line2 = CopyString(buffer);
    . . . (其余程序) . . .
}
```

另外一个解决方案就是利用在simpio.h中定义的函数ReadLine，它也可以避免与fgets相关的其他问题：

- 难以确定缓冲区的大小。有些文件中的行比较长，因此很难选择一个适合所有文件的缓冲区的大小。
- 难以判断是否超出了缓冲区的界限。如果在调用函数fgets的过程中给定一个最大值，则意味着数据不会被写到所分配的缓冲区之外，但我们同样需要知道fgets是否读入了一个完整的行。使用fgets时的唯一做法是浏览缓冲区内的所有字符，检测是否包括换行符。
- 而fgets存储换行符通常也会引起麻烦。在大多数的程序中，换行符只是一行结束的标记，而并不真正是数据的一部分。使用fgets意味着还需采取其他的步骤将缓冲区中的换行符删除。

函数ReadLine与第2章中介绍的函数GetLine很相似，唯一的区别在于其输入数据来自作为参数传递的一个数据文件。与fgets相比，ReadLine具有以下优点：

- ReadLine在需要时自动分配堆内存，使得缓冲区不可能溢出。
- ReadLine删除了标记每一行结束的换行符，所返回的数据只包括这一行中的字符。
- ReadLine返回的每一个字符串都保存在各自的内存中，因此在存储一个字符串之前不必考虑是否需要进行复制。

函数ReadLine在遇到文件结束符时返回NULL。 538

15.5 格式化I/O

在标准I/O库提供的操作中，格式化I/O函数printf和scanf最能体现C语言的特征。从第2章起我们已经开始使用函数printf，但对其功能的了解还只停留在表面上。接下来几个小节将介绍关于printf/scanf函数的更多内容，以便读者能够更有效地利用它们进行编程，以及更透彻地理解使用这两个函数的其他程序。

- 其他字符，这些字符必须与输入中的下一个字符相匹配。这使得程序能够检查所需的标点符号，如两个数字之间的逗号等。

常见错误 记住，scanf在控制字符串后的每个参数都必须是一个指针。对于简单变量来说，前面的&表明取的是它的地址。但如果该参数是一个字符数组，则该变量的名称已经自动解释为指针，因此无需再使用&。

转换说明在结构上与printf相似，但可用的选项是不同的。scanf的转换说明由以下的几种选项组成，必须遵循下面的顺序：

- 由星号（*）表示的赋值屏蔽标志，该标志表明不将输入流中的值读入参数指定的地方。
- 一个可选的数字字段宽度，指出该字段可以读入的最大的字符个数。
- 大小的说明，h表示short类型的整数值，l表示long类型的整数或double类型的浮点数。
- 转换说明符，通常为表15-1中所示的某一种说明符。

scanf的三种形式都会返回成功转换后的数据，而忽略由*号屏蔽掉的数据。如果在转换之前读到文件结束标志，则scanf返回EOF。 540

表15-1 scanf的转换说明

代 码	说 明
%d	以十进制整数的形式读入下一个值，并将该整数值存入下一个指针参数指定的存储单元。变量的大小应与说明指定的大小相匹配
%f %e %g	以浮点数的形式读入下一个值，并将该浮点数存入下一个指针参数指定的存储单元。如果指针类型的目标为float，则转换说明应为%f，如果指针类型的目标为double，则转换说明应为%lf，%e和%g与%f是相同的，它们是为了与printf函数相对应而设立的
%c	读入下一个字符，并将其存入下一个参数指定的地址，该参数必须是一个字符指针。与其他转换说明不同的是，%c在转换时不会跳过空白字符
%s	读入一个字符串，存入下一个参数指定的字符数组的连续元素中。调用程序必须确保该字符数组有足够的空间存储读入的值。遇空白字符时停止读入
%[...] %[^...]	转换说明可由一对方括号[]及其括住的字符集组成。在这种情况下，读入一个字符串并将其存入下一个参数指定的地址中，直到遇到不包含在字符集中的字符时停止读入。如果字符集由^开始，则读入字符集以外的字符。例如，转换说明为%[0123456789]，则读入由数字组成的字符串；如果转换说明为%[^.!?]，则读入的字符不包括句号、惊叹号及问号
%%	不进行任何转换，表示读入一个百分号

15.5.3 用scanf读入字符串

表15-1中包含了几个用于读入字符串的转换符——%s、%[...]和%[^...]。使用这几个转换说明时，调用程序必须负责分配足够的空间来存放数据。因此，scanf中用来存放字符串的参数应该为调用程序声明的一个数组。例如，为了读入一个由空白字符界定的字符串，必须事先声明一个数组来存放字符串，声明如下： 541

```
#define MaxWord 25
char word[MaxWord];
```

一旦获得用于存放结果的空间，程序就可以通过下面的语句读入字符串了：

```
fscanf(infile, "%s", word);
```

要注意，在变量名的前面没有&字符。函数fscanf要求它的所有参数都为指向存储空间的指

```
    if (nscan != 3 || termch != '\n') Error("Bad input line");
}
```

15.5.4 格式化I/O的一个实例

为了更好地理解scanf的工作原理，最基本的方法是研究一个使用scanf的实例，然后自己写一个类似的程序。例如，需要从文件elements.dat中读入数据，该文件为每一个化学元素列出下列信息：

- *元素名称*，通常不超过15个字符。
- *化学符号*，通常不超过2个字符。
- *原子序号*，表示原子核中质子数的一个整数。
- *原子量*，一个浮点数（用来表示该元素的各同位素的平均质量，但具体值对本例没有影响）。

543

文件的前十行如下所示：

```
Hydrogen, H, 1, 1.008
Helium, He, 2, 4.003
Lithium, Li, 3, 6.939
Beryllium, Be, 4, 9.012
Boron, B, 5, 10.811
Carbon, C, 6, 12.011
Nitrogen, N, 7, 14.007
Oxygen, O, 8, 15.999
Fluorine, F, 9, 18.998
Neon, Ne, 10, 20.183
```

假设现在要从上面的文件中读取数据，并以下列表格形式输出：

```
      Element (symbol)     Atomic Weight
-   ----------------------------------
  1. Hydrogen (H)              1.008
  2. Helium (He)               4.003
  3. Lithium (Li)              6.939
  4. Beryllium (Be)            9.012
  5. Boron (B)                10.811
  6. Carbon (C)               12.011
  7. Nitrogen (N)             14.007
  8. Oxygen (O)               15.999
  9. Fluorine (F)             18.998
 10. Neon (Ne)                20.183
```

在输出中，首先出现原子序号，然后是元素名称、化学符号，这两个内容显示在一个固定宽度的字段中，最后是原子量。

第一步是为fscanf设计一个控制字符串，以便正确地从文件中读入一行。输入行由元素名称开始，元素名称是用逗号标志结束的字符串。最简单的方法是利用转换符%15[^,]来读入第一个逗号前出现的字符。字段宽度为15保证了elementName的缓冲区不溢出。控制字符串中在此转换符后应该跟一个逗号，与输入流中的内容相对应，此后为一个空格，用来跳过逗号后面的所有空格。化学符号通过转换说明%2[^,]来读入，情况与前一个转换说明基本相同，只是字段宽度较小。最后两个字段分别为int和double型的数值。为读入这些内容，

```
    infile = fopen(ElementsFile, "r");
    if (infile == NULL) Error("Can't open %s", ElementsFile);
    printf("     Element (symbol)     Atomic Weight\n");
    printf("-----------------------------------\n");
    while (TRUE) {
        nscan = fscanf(infile, "%15[^,], %2[^,], %d, %lf%c",
                               elementName, elementSymbol,
                               &atomicNumber, &atomicWeight,
                               &termch);
        if (nscan == EOF) break;
        if (nscan != 5 || termch != '\n') {
            Error("Improper file format");
        }
        sprintf(namebuf, "%s (%s)", elementName, elementSymbol);
        printf("%3d. %-20s %8.3f\n", atomicNumber, namebuf,
                                     atomicWeight);
    }
}
```

图 15-4（续）

15.5.5 使用scanf的局限

scanf的优势在于为读取固定格式的输入提供了一个简便的方法。通常只要确定一个正确的转换说明，就可通过调用一个函数来读取很多数据项。特别是需要编写测试应用的代码并且读入一些测试数据时，scanf不失为一个很好的工具。在这种情况下，测试数据及其格式都应该是已知的。

但是，scanf函数往往对于无法控制来源的输入流束手无策。如果输入数据中可能存在错误，程序就必须测试这些输入数据，确保它具有正确的格式。但是，使用scanf时通常不可能彻底检查输入数据。在商业化的C程序中，错误检测是极其重要的一个环节，因此scanf 546
并没有得到广泛的使用。对ANSIC库进行标准化的委员会主席P.J.Plauger给出了这样的提示：

> 你会发现输入的转换说明并不像输出的转换说明那样完整。很多时候，你希望控制输入。或者你会发现不可能确定哪里出现了输入错误，并适当地恢复错误……。随时准备……在某一时刻放弃输入函数吧。多年的经验告诉我们，它们的用途实在有限[⊖]。

小结

如果想在一个程序执行结束后永久地保存信息，则必须将其存入支持永久存储的辅助存储设备（如磁盘等）上的文件中。程序设计中最常用的文件形式为文本文件，这种文件由字符数据组成。文本文件可以用来进行输入和输出，且通常按顺序进行处理。

stdio.h接口提供了多种用于输入/输出操作的函数，可根据不同需要进行选择。例如，可以逐字符地处理文件，也可以逐行处理，还可以按照一组格式化的字段进行处理。

本章的重点内容包括：

- 要在程序中使用文本文件，首先需要声明一个FILE*类型的变量来保存系统追踪对该文件进行的操作时所需的信息。

⊖ P.J.Plauger, *The Standard C Library*, Englewood Cliffs, NJ: Prentice Hall, 1992, p. 268。

程序设计练习

1. 写一个程序wc.c，要求读入一个文件，并报告该文件中有多少行、多少个字和多少个字符。在本程序中，字由连续的字符组成，以空白字符为界。例如，文件lear.txt中包含《李尔王》中的下列段落：

```
Poor naked wretches, wheresoe'er you are,
That bide the pelting of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loop'd and window'd raggedness, defend you
From seasons such as these? O, I have ta'en
Too little care of this!
```

执行程序后应该得到下面的结果：

```
File:  lear.txt↵
Lines:    6
Words:   43
Chars:  254
```

2. 出版商发现有时不必考虑文字，仅仅对文字布局及版式设计进行考察是有好处的。为了达到这个效果，出版商会对部分样张进行排版，将原来的字母替换成随机选择的字母。这样得到的文本在格式和标点上与原文完全相同，只是不再表达原来的意思。在出版业中，这样的文本替换被称为greek，有可能出自于Julius Caesar中的一句谚语“It's all Greek to me”（我对此一窍不通）。 549

写一个程序greek.c，要求能够从一个输入文件中读取字符，并对字符进行适当的随机替换，最后输出至控制台。输入中的大写字符应被任何随机的大写字符替换，而小写字符则被随机的小写字符替换。对于非字母的字符不进行替换，照原样输出。例如，如果输入文件troilus.dat是从Troilus and Cressida中节选的：

```
Ay, Greek; and that shall be divulged well
In characters as red as Mars his heart
Inflamed with Venus:
```

那么程序的输出结果应为：

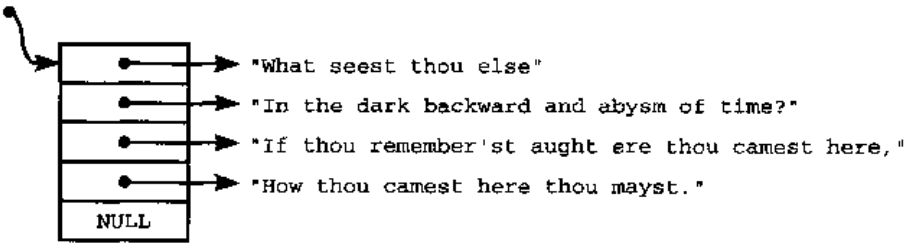
```
Hb, Jwyqt; cwq ocgs lsosn jo hdricyoc lino
Dm bzongsmdrv uj qew ya Okor umj ioyvq
Ipqpqnpj vrvy Snszv:
```

3. 一些文件运用制表符将数据按列对齐。但是对那些不支持制表符的应用来说，这种做法会导致错误。对于这些应用来讲，需要一个程序来将输入文件中的制表符转换为相应个数的空格。在程序设计中，制表符通常为8位。例如，假设输入文件中有这样一行表格：

```
abc⟶| nopqr⟶| xyz
```

```
What seest thou else
In the dark backward and abysm of time?
If thou remember'st aught ere thou camest here,
How thou camest here thou mayst.
```

如果将prospero.txt作为infile的输入打开，并调用函数ReadFile(infile)返回一个指向动态字符串数组的指针，如下面的图所示：



在设计ReadFile的实现时，最重要的一个问题是应该为动态数组分配多少空间。因为在读一个文件之前，我们无法知道其中包含有多少行的内容。解决这个问题最常用的方法是为数组任意指定一个初始大小，之后如有必要再随时增加新的空间。如果需要分配更多空间，需要按照以下几步进行：

a. 分配一个新的动态数组，该数组要比原来的数组大。

b. 将旧数组的内容复制到新的数组中。

c. 释放旧的数组。

d. 将新数组的指针赋给指向数组的变量。

将ReadFile的实现写入一个单独的readfile.c文件中，并设计一个readfile.h接口，使客户可以调用这个接口。用readfile.h接口写一个主程序将文件中的行按倒序输出。例如，以prospero.txt中的内容作为输入，则程序的输出应为：

```
How thou camest here thou mayst.
If thou remember'st aught ere thou camest here,
In the dark backward and abysm of time?
What seest thou else
```

552

6. 在20世纪60年代，演员Steve Allen经常在他的喜剧表演中玩一个名为Madlibs的游戏。Allen让观众提供某一类型的词，如动词、形容词、或复数名词等，并将这些词填入事先准备好的文本的空格中，再将它们念给观众听。

为了说明这一过程，假设事先准备好的文章是从《哈姆雷特》中节选的，并已将几个关键字删去了，如下所示：

To (动词) or not to (也是动词): that is the question.

Whether 'tis nobler in the mind to suffer

The slings and arrows of (形容词) fortune,

Or to take arms against a sea of (名词复数),

And by opposing end them?

观众选择适当类型的词将空格填满，但事先并不知道原文是什么。按照所选的词，得到结果可能是相当可笑的，尽管形容词silly可能用的很多。

注意，程序将单词中的所有字符都转换为小写，因此Some和some不会被作为不同的词处理。

在设计解决方案时，要特别考虑模块的分解。什么样的模块有助于解决这个问题？特别是有没有现有的模块可以应用？

8. 写一个程序，将包含你所要购置计算机设备的清单的文件转换成更优美的格式。输入文件由若干行信息组成，每一行有以下几个字段：

a. *零件编号* 最多由六个字符组成，从每一行的开始至第一个空格处为止。 554

b. *产品名称* 最多由20个字符组成。这一字段从零件编号后开始，一直到第一个“/”字符为止，该字符必须出现在每行中。

c. *产品数量* 为一整型值。

d. *单位价格* 一个浮点型数值，通过@符号与产品数量进行区分。

在字段与字段之间可出现空白字符。但每一行中都要包含这四个字段的数据以及所需的区分标志。输入文件的格式如下面的文件order.dat所示：

```
NC271X Notebook computer / 1 @ 1729.00
LP552 Laser printer / 1 @ 1499.00
TC552V Toner cartridge / 2 @ 99.00
DS420 3.5" diskettes / 10 @ 2.99
CC542 Carrying case / 1 @ 27.99
```

程序应该读入该输入文件，并以固定宽度显示每个字段。另外，程序需计算出每一行货品的总价（产品数量×单位价格），以及整个订单的总额。按照上面给出的数据，程序的结果应如下所示：

```
Order file: order.dat↵
NC271X    Notebook computer      1 @ 1729.00 = 1729.00
LP552     Laser printer          1 @ 1499.00 = 1499.00
TC552V    Toner cartridge        2 @   99.00 =  198.00
DS420     3.5" diskettes        10 @    2.99 =   29.90
CC542     Carrying case          1 @   27.99 =   27.99
-------------------------------------------------------
TOTAL                                          3483.89
```

9. 利用第7章的图形库，写一个程序在屏幕上显示多边形。多边形顶点的坐标保存在一个数据文件中，每一行为一对坐标。每一个坐标由两个浮点型数值组成，由圆括号括起，中间以逗号分隔。输入文件中可能包含多个多边形的顶点坐标，每一个多边形靠一个空白行与前面一个多边形区分开来。例如，文件polygon.dat包含了一个正方形坐标和一个三角形的顶点坐标，如下所示： 555

```
(0.5, 0.5)
(1.0, 0.5)
(1.0, 1.0)
(0.5, 1.0)

(2.0, 0.5)
(3.0, 0.5)
(2.0, 1.0)
```

第16章 记　录

Let's look at the record.

— Al Smith, presidential campaign slogan, 1928

学习目标

- 理解记录的概念以及在编程中的重要作用。
- 理解定义一个记录类型和声明一个记录变量的差别。
- 能够通过“.”和“→”运算符来操控记录以及记录指针。
- 能够为新记录动态地分配存储区。
- 理解数据库的设计过程。

557

在第11章学习数组的时候，我们已经初步理解了计算机程序设计的一个重要理念：利用复合的数据结构表示复杂的信息集合。在程序中声明一个数组，即可将任意多的数据聚集起来构成一个概念上的整体。需要时可以从数组中选出元素，并单独进行操作；也可以将它们作为一个整体，同时进行操作。

现代程序设计语言的一大特点是能够将各自独立的数据组织成一个彼此相关的整体。过程和函数可以将不同的操作统一在一个名称下。在数据处理方面，复合的数据结构，例如数组，可以实现这种统一。在很多情况下，将小的程序段整合成为单一的、更高级的结构，有助于简化程序并提高程序的可读性。这种统一不是最近才出现的，这一理念已经对国家和社会行为产生了影响，如劳工赞歌中的“团结让我们更有力”，以及美国国玺上的名言“一从众中来”等。

当为现实世界中的一组数据建模时，数组是非常有用的一个工具。而这样的数据往往具备两条基本性质：首先，数据必须是有序的，因而可以通过索引号码来引用某个元素。其次，数据必须为同质的，即所有的元素的基本类型必须相同。当试图对现实世界中的一组或一系列相似的事物进行建模的时候，数组往往是最理想的工具。另一方面，将一组无序的、异质的数据看作一个整体也很重要。在C语言中，这样的一组数据被称为**结构**（structure），而在计算机科学这一领域内则被称为**记录**（record）。本书在大多数情况下使用记录一词，但需要突出C语言中对于记录的特殊实现时会改用结构一词。

16.1 数据记录的概念

为了更好地理解记录的概念，假设你负责某家小公司的薪金支付系统。你必须了解每一个雇员的信息。例如，为打印工资单，你必须知道雇员的姓名、职位、社会保障号、薪水、扣缴税款的情况，以及其他相关信息。这些信息组成了一个雇员的记录。

雇员记录是什么样子的呢？把记录想象成表格中的行是最容易理解的。例如，想象一下查尔斯·狄更斯的小说《圣诞颂歌》中斯克鲁齐（Scrooge）和马列（Marley）的小公司，并将今天的社会保障号码及扣缴税款补贴加进去，则雇员花名册包含两个记录，相关的数据如下所示：

558

声明和函数中的变量声明类似。例如，下列代码定义了名为employeeT的结构类型，用以表示雇员记录：

```
typedef struct {
    string name;
    string title;
    string ssnum;
    double salary;
    int withholding;
} employeeT;
```

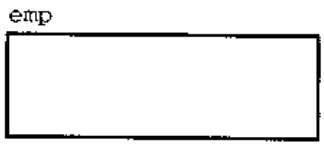
这一定义为具有新类型employeeT的所有对象提供了一个模板。每一个对象都具有五个字段，由name字段开始（该字段为字符串），直至withholding字段结束（该字段为整型数）。

16.2.2 声明结构变量

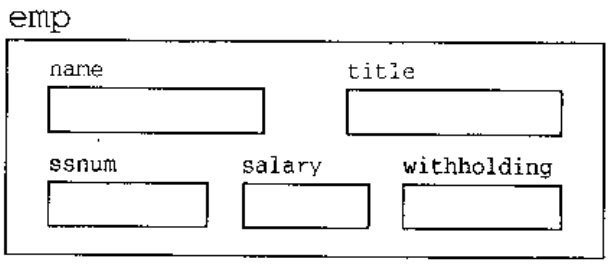
定义了一个新类型后，下面的一步就是声明该类型的变量。例如，有了employeeT这个类型，就可以通过下列代码定义该类型的一个变量emp：

```
employeeT emp;
```

如果需要用方框图表示这个变量，可以采用两种方式。如果从整体的角度来看待，即在概念上从一个较远的距离来观察，则得到右图所示的一个名为emp的方框。但如果从近处看细节，会发现标签为emp的方框内部还存在着五个单独的方框（见下图）：



560



16.2.3 记录选择

当通过如下语句声明了变量emp后，

```
employeeT emp;
```

就可以用这个名称指代该记录，但有时需要打开该记录结构并对单个字段进行操作。如果需要引用记录中的某一个字段，需要写下整个记录的名称，后面跟符号“.”以及该字段的名称。因此，为引用存储在emp中雇员的职位，需要写以下代码：

```
emp.title
```

在此，符号“.”称为“点”，因此可以将这个表达式读作“emp点title”。通过点运算符选择字段被称为**记录选择**（record selection）。

16.2.4 记录初始化

和对待其他类型的变量一样，我们可以通过赋值的方式对记录变量进行初始化。点运算符返回一个左值，表示可以对一个记录选择表达式进行赋值。例如，如果执行下列语句：

```
emp.name = "Ebenezer Scrooge";
emp.title = "Partner";
```

```
pointT AddPoint(pointT p1, pointT p2)
{
    pointT p;

    p.x = p1.x + p2.x;
    p.y = p1.y + p2.y;
    return (p);
}
```

当调用函数AddPoint(p1, p2)时，(其中p1，p2都为pointT类型的值)，函数返回的点的坐标是p1和p2相应坐标的和。

当调用参数为记录的函数时，记录参数的值就复制到相应的实参。当函数改变实参的值，或改变某个字段内容时，形参还保留着原来的值。这种处理方式和C语言处理所有参数（除数组以外）一样。

16.3 数组与记录的结合

现代程序设计语言的一大特点是，当你拥有一个类型时，就可以利用它来创建更复杂的新类型。为了达到这一目的，通常可以定义记录数组，或者包含数组的记录。同理，这样的新类型同样可以用来创建更加复杂的类型。

例如，在前面的16.2.1节中定义了一个名为employeeT的类型。一旦定义了该类型，就可以用它来声明数组，而数组的元素的类型就是employeeT。下面的代码

```
int scores[10];
```

声明scores是具有10个整型值的数组，而

```
employeeT staff[10];
```

563

就声明staff是具有10个雇员的数组。雇员Ebenezer Scrooge和Bob Cratchit的记录刚好适用于这个数组，成为staff数组的前两个元素，如下所示：

staff

0	Ebenezer Scrooge	Partner	271-82-8183	250.00	1
1	Bob Cratchit	Clerk	314-15-9265	15.00	7

可以选取staff的任意元素，就如从其他数组中选取元素一样。例如，可以用staff[1]来选取Bob Cratchit的整个记录：

Bob Cratchit	Clerk	314-15-9265	15.00	7

对于这条记录，可以进一步选取其中的字段，例如：

```
staff[1].name
```

就选取了Bob Cratchit的名字：

Bob Cratchit

选取名字字段可能已经满足了应用的需要。但也可以进一步进行选取。Name字段的值是一个字符串，因此可以继续对其中的元素进行选取。如果想知道staff数组中第一个雇员的名字的首字母，就可以用下面的语句：

```
staff[1].name[0]
```

你还可以定义适合在雇员数据库系统中使用的其他函数。

16.4 记录的指针

虽然在C语言中会直接使用一些简单的记录，但大多数情况下，存储结构化数据的变量通常被定义为记录的指针，而不是记录本身。指向记录的指针通常比记录小且较容易操作。更重要的是，当把记录的指针传递给某个过程时，过程可以改变记录的内容。如果直接传递记录，该记录将被复制（如同复制一个整型值一样），这样就不可能对其内容作出永久性的修改。在第11章中我们已经认识到，在C语言中，利用指针传递数组变量可在函数中对数组元素进行修改，同样的，传递记录的指针可以对记录中的字段进行操作。

为了练习使用记录指针，可从声明指向employeeT记录的指针入手，employeeT的定义如下：

```
typedef struct {
    string name;
    string title;
    string ssnum;
    double salary;
    int withholding;
} employeeT;
```

有了这个定义，就可以声明一个employeeT类型的指针变量，代码如下：

```
employeeT *empptr;
```

该语句说明变量empptr是一个指向employeeT类型对象的指针。进行声明后，仅仅为指针提供了空间。在使用指针empptr前，仍然需要为记录中的字段提供完整的空间。

在介绍如何为记录创建空间之前，有一个问题值得我们考虑。在前面提到的例子中，我们定义了一个代表员工的记录类型，并声明了指向该类型的指针。这样的操作让我们必须同时考虑两个不同的类型：employeeT类型，以及employeeT类型的指针。如果能够单独采用一个具有指针优点的简单类型，那么我们的操作会变得更容易。

16.4.1 定义一个指向记录类型的指针

在前面的一节中提到，employeeT类型被定义为一个记录，因此在声明该类型的指针时前面必须加一个*。在大多数情况下，可以采用另外一种声明策略。如果你希望通过地址访问记录类型的值，可以将employeeT定义为一个指针而不是一个记录。我们只需对前面employeeT的定义做出一个微小的改动，即可达到这一目的。如下所示： 566

```
typedef struct {
    string name;
    string title;
    string ssnum;
    double salary;
    int withholding;
} *employeeT;
```

此类型定义与前面的定义的唯一差别就在于employeeT前面的星号。这个定义表明employeeT类型为指向所说明记录的指针，虽然在定义中说明的记录没有一个明确的名称。

在建立了这个新类型定义之后，就可以用与前面类似的方法来声明employeeT的对象了，声明如下：

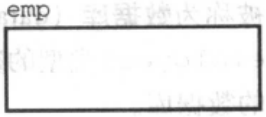
```
employeeT emp;
```

虽然使用GetBlock和sizeof的习语简单明了，但在基于记录的程序设计中为新记录分配空间的操作是十分基本的操作，因此需要更简单的方式。基于这种考虑，genlib.h接口定义了函数New，该函数以指针类型名作为参数，并返回一个指针指向足够大的空间来存储可能的数值。因此，如果有下列语句：

```
emp = New (employeeT);
```

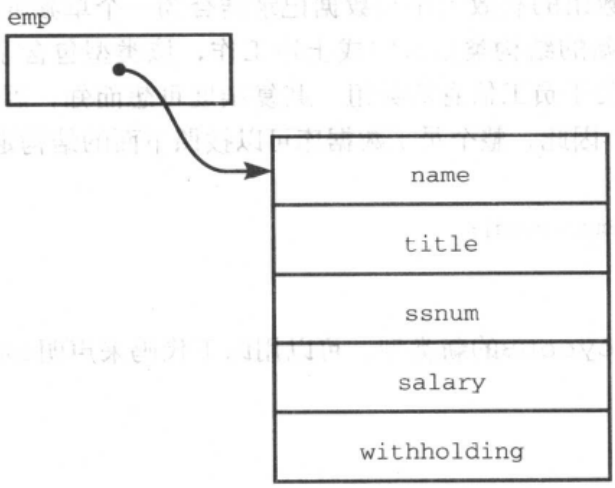
则系统会从堆中为一个员工记录分配空间，并返回一个指向该记录的指针。

这一操作的结果可以通过下面的描述来说明。在执行对New的调用前，emp中存储了一个指针，但却并未进行初始化。因而变量emp与任何记录的存储都没有关联，可以用一个空的方框表示（如右图所示）。赋值语句




```
emp = New (employeeT);
```

为记录分配了空间，并将指针的地址赋给emp，这一过程可以用下面的图表示：



16.4.3 对记录指针进行操作

虽然大多数情况下会把emp这样的变量作为一个完整的实体考虑，但有些时候我们也需要访问记录中的某个字段。在没有使用指针的记录中，我们使用点运算符来选取某个字段。当使用记录指针时，应该怎样访问某个字段呢？

在寻求答案的过程中，我们很容易被直觉误导。对于下面的声明

```
employeeT emp;
```

下面的语句是不正确的：



```
*emp.salary
```

（这是不正确的）

569

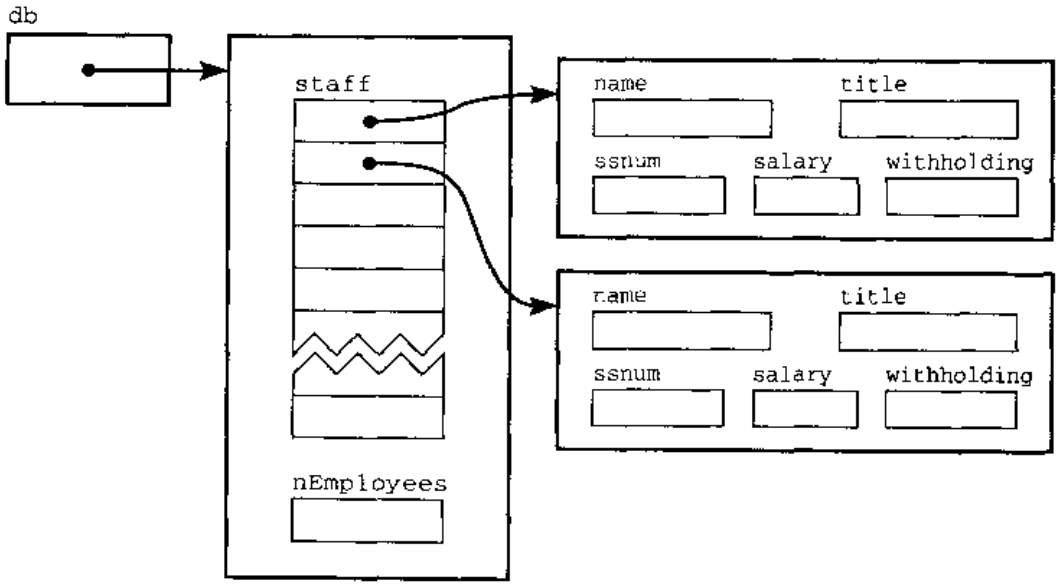
与我们的设想相反，这条语句不能选取emp所指对象的salary字段，因为这条语句中的运算符优先级不支持这样的数据解释。选择运算符的优先级高于间接引用*，因此该语句被解释为如下所示的无意义的表达式：

```
*(emp.salary)
```

而不是原本打算的：

```
(*emp).salary
```

最后，staff数组中的每一个元素（根据最近的employeeT类型定义）都是一个指向记录的指针，因此完整的图应该包含指向单独employeeT结构的指针，如下图所示： 571



为了创建这个结构，应该从部分着手。对于结构中的每一个由指针指向的方框，都需要调用函数New来为该记录分配存储空间。

为使这个例子更加具体，假设我们需要编写程序从用户处读入一个员工信息数据库。对于每一个记录，程序需要得到每个雇员五方面的信息。需要用一个空白的name字段作为数据库结束的标记。每个记录都应该存入数据库中，且要对记录进行计数。

读入数据库的操作很复杂，应该定义为一个函数。由于数据库为一个指针，ReadEmployeeDatabase只能分配指针需要的存储空间，并返回一个指针。因此，主程序声明一个数据库：

```
employeeDB db;
```

并通过调用以下函数读入值：

```
db = ReadEmployeeDatabase( );
```

ReadEmployeeDatabase的实现很简单，如果用逐步精化的方法就更简单。只需将其分解为两个函数：函数ReadEmployeeDatabase创建资料库结构；函数ReadOneEmployee读入每个员工的信息。由于函数ReadOneEmployee负责从文件中读入数据，因此该函数必须能够判断文件是否结束，并将信息通过返回一个特殊的标志传递给ReadEmployeeDatabase。由于函数ReadOneEmployee返回的值为employeeT类型的指针，因此最好用NULL作为文件结束标志。 572

函数ReadEmployeeDatabase的代码为：

```
static employeeDB ReadEmployeeDatabase(void)
{
    employeeDB db;
    employeeT emp;
    int nEmployees;

    db = New(employeeDB);
    nEmployees = 0;
```

```
            db->staff[i]->salary *= 2;
        }
    }
}
```

16.6 基于记录的应用程序设计

二十年前，当计算机刚刚作为通用工具进行普及时，在教育界掀起了一股热潮，即让计算机参与到教学的过程中。有人提出了一种技术，称为**程序化教学**（programmed instruction），即由计算机辅助的教学工具提出一系列的问题，并由前面问题的回答决定后面的问题。如果学生的回答全部正确，则计算机会跳过大部分简单的问题，直接提出难度较大的问题。而对在回答中遇到困难的学生，整个过程会进行的较慢，留出时间让学生复习总结。

设想一下，如果要求你编写一个应用程序实现程序化教学。简单来说，你的程序必须能够做到下面几点：

- 向学生提出问题。
- 从学生处获得答案。
- 继续下一个问题，根据学生的答案选择下一个问题。

这样的程序比商业化的应用程序简单很多，但从中可以体验程序设计所涉及到的基本原理。 574

16.6.1 使用数据库的重要性

将程序化教学的应用程序设计为一个过程的集合是可行的。每个过程提出一个问题，读取学生的答案，并根据学生的答案调用另外一个适当的过程。但是这样的程序很难进行改变，如果想加入新的问题或为另外一门课程设计程序，则需要重新编写过程。对于懂得编程的人来讲，写程序并不是很困难的事，但并不是所有人都懂得编程。大多数程序化教学的课程都是由某一学科的教师进行设计，而这些教师通常不是程序员，让他们来编写程序会限制应用程序的功能。

作为该项目的程序员，你的目标是开发一个应用程序，提供程序化教学，使不具备编程技能的老师可以提出问题、接受答案并根据前后对照的信息使程序按照适当的顺序进行。为了达到这个目标，可以将程序设计为一个通用的工具，该工具可以从文件中读入与程序化教学相关的信息和数据。如果你采用这种方法，只要提供不同的数据文件，同一个程序就可以支持不同的课程了。

在该应用程序的设计中，可以从以下几个具有普遍性的问题入手进行考虑，例如：

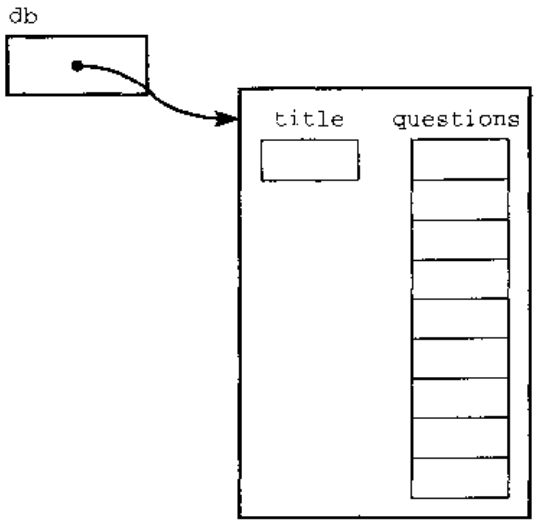
- 一般问题的要求是什么？要特别理解程序所必须支持的所有操作，而不仅仅是某一学科的要求。
- 如何在你的程序中表示程序化教学课程所需的信息？在设计阶段，你需要开发一个合适数据结构，它可以是记录和数组的结合。
- 课程的数据文件是怎样的？当你做出决定时，你必须意识到数据文件是由非程序员编辑操作的，他们的专长在于所教授的课程，而不是编程。
- 如何将数据文件使用的外部表示转换为内部表示？
- 如何编写程序对数据库进行操作？

本章的剩余部分将依次探讨这几个问题。

16.6.2 问题的框架

在某一层次上，写出程序的提纲并不困难。当程序运行时，基本的操作是重复执行以下

解应该如右图所示。



虽然现在仍然不知道用什么类型来表示问题，但仍然可以将这个图转换为一个结构定义。我们已经知道了其他字段的类型，因此可以写出一个恰当的记录定义，如：

```
typedef struct {
    string title;
    questionT questions[MaxQuestions+1];
} *courseDB;
```

常量MaxQuestions为程序所允许的问题数目的最大值。由于数组下标总是从0开始，而课程总是从问题1开始，因此需要为数组增加一个元素，使得第MaxQuestions个问题可以存在。

courseDB的定义使我们可以声明变量db来存放整个数据库，定义如下：

```
courseDB db;
```

但是这个声明仅为指针保留存储空间，因此是与前面所示的虚线框部分相应的。当在程序中创建结构时，需要为该结构分配空间，代码如下：

```
db = New(courseDB);
```

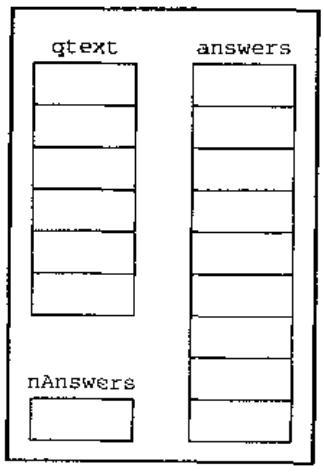
此时，需要增补一些丢失的细节。例如，我们仍然没有为courseDB定义中的类型questionT作出定义。为使数据结构完整，还需补充这方面的设计。问题由文本组成，其中包括多行文件以及一系列可能的答案。这两个结构都可由数组表示。问题的文本是字符串的数组，每一个字符串包含了一行的内容。而答案存储在一个结构较为复杂的数组中。目前，我们可以将答案声明为一个类型answerT的数组，其细节将在下一步完善。

通常情况下，我们需要提供一个机制来追踪数组的有效大小，因为它通常比所分配的空间小一些。记录有效大小可以利用下面两种方式：

- 可以在数组的最后一个值后面增加一个标记值。
- 可以将元素的个数存入一个整型变量，并将其作为记录的一部分。

究竟采用哪种方式取决于数组的使用方式，以及是否可以选择一个合适的标记值。

为了介绍这两种模式的操作，示例程序采用了两种定义questionT的方式。组成问题文本的行存入了一个由字符串组成的数组。对于指针数据来讲，常量NULL是一个最佳的标记值。而由于答案数组的元素都是answerT类型的，这种类型仍未进行定义，因此很难知道选择什么样的标记值。因此，在此例中更适合用一个变量存储答案的数量。



一个问题的结构可用右图表示。因为这个结构比较复杂，而且需要将其考虑为一个整体，因此可以将此结构定义为一个指针类型，如下所示：

```
typedef struct {
    string qtext[MaxLinesPerQuestion+1];
    answerT answers[MaxAnswersPerQuestion];
    int nAnswers;
} *questionT;
```

将此结构声明为一个指针意味着必须调用New来为每个问题分配

问题的文本由单独的一行文件构成，接下来是两个可能的选择。如果用户输入false，则程序继续提出问题2；而如果用户输入true，则程序跳至问题3。

这个例子提出了一个有趣的问题，即如何将问题序号与每一个问题联系起来？一种方法是将文件中的问题排列并依次编上序号。这种做法给程序员带来便利，却给课程设计者造成一定的困难。

如何理解这一点呢？假设课程设计者想在现有课程的开始处加入一个问题。那么，所有后续的问题都要向后移一位，所有的问题序号也要随之改变，因此课程设计者需要花很多的时间完成后续的处理工作。更好的办法是由程序编写者给每个问题编一个序号。例如，如果示例中关于地球和太阳的问题是1号的话，它的文件可以包含该问题的编号，如下图所示：

```
1
True or false: The earth revolves around the sun.
-----
true: 2
false: 3
```

由课程设计者提供问题序号的优点在于，这样会使课程设计的编辑变得相对简单。如果想要增加一个问题，只需给该问题一个从未使用过的编号就可以了。其他问题的编号无需改动。课程设计者可以在文件中的任何位置加入新的问题，因为已经无需确保问题序号的连续性了。

16.6.5 程序代码

决定了内部数据结构及外部文件格式后，编写程序的过程就变得很简单了，只需将整个任务分解成单独的函数，并对它们进行逐步精化就可以了。完整的程序在文件teach.c中，如图16-1所示。 581

```
/*
 * File: teach.c
 * -------------
 * This program executes a simple programmed instruction course.
 * The course is specified by a data file containing all the
 * course information.  The data structures and the format of
 * the data file are described in Chapter 16.
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/*
 * Constants
 * ---------
 * MaxQuestions          -- Maximum question number
 * MaxLinesPerQuestion   -- Maximum number of lines per question
```

图16-1 teach.c

```
static void ReadQuestionText(FILE *infile, questionT q);
static void ReadAnswers(FILE *infile, questionT q);
static FILE *OpenUserFile(string prompt, string mode);
static void ProcessCourse(courseDB course);
static void AskQuestion(questionT q);
static int FindAnswer(string ans, questionT q);

/* Main program */

main()
{
    courseDB course;

    course = ReadDataBase();
    ProcessCourse(course);
}

/* Section 1 -- Functions to read the data file */

/*
 * Function: ReadDataBase
 * Usage: ReadDataBase();
 * ----------------------
 * This function asks the user for a file name and reads
 * in the database for the course.  The file is formatted
 * as discussed in the section "Designing the external
 * structure" in Chapter 16.
 */

static courseDB ReadDataBase(void)
{
    FILE *infile;
    courseDB course;

    infile = OpenUserFile("Enter name of course: ", "r");
    course = New(courseDB);
    course->title = ReadLine(infile);
    while (ReadOneQuestion(infile, course));
    fclose(infile);
    return (course);
}

/*
 * Function: ReadOneQuestion
 * Usage: while (ReadOneQuestion(infile, course));
 * -----------------------------------------------
 * This function reads in a single question from infile into the
 * course data structure.  As long as the complete question is
 * read successfully, this function returns TRUE.  When the end
 * of the file is encountered, the function returns FALSE.
 * Thus, the "Usage" line above reads the entire data file.
 */

static bool ReadOneQuestion(FILE *infile, courseDB course)
{
    questionT question;
```

583

图 16-1 (续)

```
static void ReadAnswers(FILE *infile, questionT q)
{
    string line, ans;
    int len, cpos, nextq, nAnswers;

    nAnswers = 0;
    while ((line = ReadLine(infile)) != NULL
           && (len = StringLength(line)) != 0) {
       cpos = FindChar(':', line, 0);
       if (cpos == -1) Error("Illegal answer format");
       ans = SubString(line, 0, cpos - 1);
       nextq = StringToInteger(SubString(line, cpos+1, len-1));
       q->answers[nAnswers].ans = ConvertToUpperCase(ans);
       q->answers[nAnswers].nextq = nextq;
       nAnswers++;
    }
    q->nAnswers = nAnswers;
}

/*
 * Function: OpenUserFile
 * Usage: fileptr = OpenUserFile(prompt, mode);
 * --------------------------------------------
 * This function prompts the user for a file name using the
 * prompt string supplied by the user and then attempts to
 * open that file with the specified mode.  If the file is
 * opened successfully, OpenUserFile returns the appropriate
 * file pointer.  If the open operation fails, the user is
 * informed of the failure and given an opportunity to enter
 * another file name.
 */

static FILE *OpenUserFile(string prompt, string mode)
{
    string filename;
    FILE *result;

    while (TRUE) {
       printf("%s", prompt);
       filename = GetLine();
       result = fopen(filename, mode);
       if (result != NULL) break;
       printf("Can't open the file \"%s\"\n", filename);
    }
    return (result);
}

/* Section 2 -- Functions to process the course */

/*
 * Function: ProcessCourse
 * Usage: ProcessCourse(course);
 * -----------------------------
 * This function processes the course supplied by the caller.
 * The basic operation consists of a loop that (a) prints out
 * the current question, (b) reads in an answer, (c) looks up
```

586

图 16-1 (续)

```
static int FindAnswer(string ans, questionT q)
{
    int i;

    for (i = 0; i < q->nAnswers; i++) {
        if (StringEqual(ans, q->answers[i].ans)) return (i);
    }
    return (-1);
}
```

图 16-1 (续)

这个程序值得我们花时间仔细阅读，并注意下面几个问题：

1) 程序是怎样利用文件中的数据创建内部数据结构的?

2) 程序如何运用这些数据结构处理每个问题?

16.6.6 数据驱动设计方法的重要性

图16-1中所示的程序teach.c从课程数据文件中接收了所需的全部数据。该应用程序所提出的问题，接受的答案，以及问题的顺序等信息全部来自文件，而不是程序本身。根据从数据库中获得信息而运行的程序称为**数据驱动**（data-driven）程序。与将所有信息包含在程序设计过程中的程序相比，数据驱动的程序通常较短、较灵活，且较容易维护。

我们只需运行程序就可以理解teach.c这样的数据驱动系统是多么的灵活。按照这个项目的最初目标，程序teach.c应该用于传统的程序化教学，如文件cs.dat最开始的几个问 588
题可参见图16-2。当程序teach.c与这个文件结合应用时，得到的结果可参见图16-2。

由于这是一个数据驱动的程序，所以同样一个程序可应用于不同的环境中。例如，图16-3中显示了数据文件advent.dat中开始的几行，这是从Willie Crowther在20世纪70年代初开发的计算机游戏改编的。当运行示例程序时，将程序teach.c与文件advent.dat结合起来时，该程序的目的与运行cs.dat时大相径庭。虽然程序teach.c没有任何改变，但程序化教学的课程已经变为了一个游戏，而唯一的差别仅在于数据文件不同。

小结

在第11章里曾经介绍过可以用数组存储同质的、有序的数据集合。本章介绍了一个同样重要的概念——记录，它可以将无序的、异质的数据存储在一个集合中。数组通常用于为真实世界中的对象集合建模，而记录则用来为由若干个可以识别的部分组成的对象建模。同时，可以将数组和记录结合起来，组成具有任意复杂度的数据层次结构。用来表示数据集合的一个完整结构称为数据库，特别当它作为数据文件表示时更是如此。

本章的重点包括以下几点：

- 一个记录是由多个组成部分组成的，而这些组成部分称为字段。在C语言中，记录常被称为结构，其组成部分被称为成员。
- 声明一个记录分为两个步骤。首先要定义类型，为记录提供模板，之后即可声明该类型的变量。
- 给定一个记录，可以使用点运算符来选取个别字段。
- 在很多情况下，使用记录指针比记录本身更合适。从空间上讲，使用指针效率更高，而且可以通过函数修改记录的内容。

```
Welcome to ADVENTURE!
1
You are standing at the end of a road before a
small brick building.  A small stream flows out
of the building and down a gully to the south.
A road runs up a small hill to the west.
-----
south: 2
north: 8
in:    8
west:  9

2
You are in a valley in the forest beside a stream
tumbling along a rocky bed.  The stream is flowing
to the south.
-----
south: 3
down:  3
north: 1

3
At your feet all the water of the stream splashes
down a two-inch slit in the rock.  To the south,
the streambed is bare rock.
-----
north: 2
south: 4
down:  4

. . .              其他的问题              . . .
```

```
Enter name of course: advent.dat↵
Welcome to ADVENTURE!
You are standing at the end of a road before a
small brick building.  A small stream flows out
of the building and down a gully to the south.
A road runs up a small hill to the west.
south↵
You are in a valley in the forest beside a stream
tumbling along a rocky bed.  The stream is flowing
to the south.
down↵
At your feet all the water of the stream splashes
down a two-inch slit in the rock.  To the south,
the streambed is bare rock.
```

图16-3 数据文件advent.dat和相关运行示例

589 ~ 591

复习题

1. 给出记录和字段的定义。当在C语言中描述这两个概念时通常用什么术语？
2. 判断题：数组中的每个元素类型都相同。
3. 判断题：记录中每个字段的类型都必不相同。
4. 声明一个记录变量的必要步骤是什么？

```
+-------------------------------------------------+
| Scrooge and Marley Ltd.                         |
|                                                 |
| Pay to the order of: Bob Cratchit       13.00   |
|                                                 |
|                                                 |
| 15.00 gross - 2.00 tax              E. Scrooge  |
+-------------------------------------------------+
```

593

你的工作包括打印支票的外框，并对内部信息进行格式化。

2. 假设需要你为图书馆建立一个计算机处理的卡片目录系统。第一步，你需要建立一个原型，使之能够存储1000本书的如下信息：

- 书名
- 一个最多包含五位作者的名单
- 美国国会图书馆的目录号
- 至少五个主题词
- 出版商
- 出版年
- 该书是否流通

为此图书馆数据库原型设计合适的数据结构，可以存储所有必要的信息。给出你的定义后，可以按照语句：

```
libraryDB libdata;
```

进行声明，且变量libdata应该包含1000册书的信息。要记住，书的实际数量通常少于这个上界。

编写过程SearchBySubject，将图书馆数据库和一个主题词作为参数。如果某本书的主题词中包含该主题词，则SearchBySubject应显示该书的书名、第一作者的姓名，以及国会图书馆目录号码。

3. 编写函数Midpoint(p1, p2)，返回p1和p2之间线段的中点。函数的参数及结果都应该为pointT类型，其定义如下：

```
typedef struct {
    double x, y;
} pointT;
```

4. 设计接口ptgraph.h，用点代替坐标表示相对位移。例如，ptgraph.h的用户调用MovePenToPoint(pt)，而不是MovePen(x,y)，其中参数pt的类型为上题中定义的pointT类型。有些函数，如InitGraphics和DrawArc不接受坐标值作为参数，对于这类函数，在你的接口中不需重新定义。实现ptgraph.h接口，使之成为graphics.h顶部的分层抽象。

5. 可以用两个整数的商表示的数称为**有理数**（rational number）。因此1.25是一个有理数，它等于5除以4。很多数不是有理数，比如π和2的平方根。在计算时，使用有理数比使用浮点数更有优势：有理数是精确的，而浮点数不是。因此，设计一个专门处理有理数的程序是非常有用的。 594

设计一个接口rational.h，允许对有理数进行简单的操作。至少，你的接口应该可以

每一行都有某种货币的名称，后跟至少一个空格，以及一单位该货币相当的美元数。因此，示例输入文件告诉我们1英镑折合$1.96，而1德国马克折合68美分。(注意文件中有一行是美元的信息，因此它的汇率永远是1.00。这行的存在可以使美元不必作为特例处理。)

编写一个程序执行以下步骤：

a. 从exchange.dat文件中读入一个合适的内部数据结构。

b. 让用户输入两种货币的名称：将前者转换，而返回后者。

c. 让用户输入原货币的一个值。

d. 将得到的值以第二货币的形式显示。最简便的方法是，将原值转换为相应的美元数，再将此美元数转换为第二种货币数。

下面是运用exchange.dat文件运行的一个实例：

```
Convert from: mark↵
Into: yen↵
How many units of type mark? 200↵
200 mark = 17435.9 yen
```

8. 在很多应用程序中，在一个字符串与其定义之间建立起类似字典似的联系非常重要。在程序中的某一点可以对一个字符串进行定义；在之后的某一点，可以查看该字符串的定义。更正规的情况是设计一个程序，将一个值和一个关键字联系起来（Define操作），然后查找与该关键字相关的值（Lookup操作）。

图16-4中所示的接口dict.h中具有这种功能，通过调用Define(key,value)，你就可以在程序后面的某一点通过调用Lookup(key)来查找该值。运用记录数组来表示数据，实现接口dict.h。

```
/*
 * File: dict.h
 * ------------
 * This interface exports functions for defining and looking
 * up words in a dictionary. The dictionary is maintained in
 * static storage private to the implementation.
 */

#ifndef _dict_h
#define _dict_h

#include "genlib.h"

/*
 * Function: InitDictionary
 * Usage: InitDictionary();
 * ------------------------
 * This function initializes the dictionary to be empty and must be
 * called before any of the other operations are used.
 */

void InitDictionary(void);
```

图16-4 dict.h

```
Input file: formlet.txt↵
   name: Dr. Roberts↵
   topic: the misuse of computers↵
   message type: e-mail message↵

Dear Dr. Roberts:

Thank you for taking the time to express your interest in
the misuse of computers. Since being elected to Congress,
the misuse of computers has been my top priority.

Because your e-mail message means so much to me, I have
taken the time to respond personally. I hear from many
constituents every day, but your e-mail message on
the misuse of computers has moved me deeply.

Please be assured that I will do all I can to fight the
special interests that oppose us on this critical issue.

                                   Sincerely yours,

                                   Your Congressperson
```

597～599

对程序Madlibs进行扩展，存储需替换字符的字典，并在每一次出现时用恰当的值替换。

10. 图16-1所示的程序teach.c有时会产生混乱的输出结果，因为在它的设计过程中有个小问题。按照图16-2中所示的课程数据库文件，学生从程序中得到的反馈必定和下一个问题相关联。如文件cs.dat中的问题3：

```
No. Floating-point numbers have fractional parts;
integers do not.
True or false: Integers can be negative.
```

前两行为前面一个问题的答案，而最后一行是下一个问题。

这两条信息的结合必定会导致错误。例如，如果学生对于问题3的回答是不可确认的，计算机将重复这个问题。但是，问题的文本中包含的对前面问题的反馈信息就已经毫无意义了。因此，如果学生做出诸如“maybe”这样的回答，屏幕上将出现这样的显示：

```
True or false: Integers can be negative.
maybe↵
I don't understand that.
No. Floating-point numbers have fractional parts;
integers do not.
True or false: Integers can be negative.
```

考虑一下如何重新设计程序teach.c来避免这一问题，实现并测试你的新设计。 600

17.1.1 递归的简单说明

为了更好地了解什么是递归，我们举一个例子，假设你被一家大型慈善机构任命为融资协调者，你的工作是筹集1 000 000美元的资金以支付该机构的运作开销。

如果你能找到一个人愿意掏出1 000 000美元，你的工作就很简单了。然而，你很难一下子找到这么慷慨大方的人。所以，你很有可能需要募集很多小笔的捐款来凑齐1 000 000美元。如果平均每笔捐款额为100美元，你可以用另一种方法完成这项工作：找10 000个朋友说服他们每个人捐出100美元。但是你又不大可能有10 000个朋友，那你该怎么办呢？

当你面对的任务超过你个人的能力时，完成这项任务的办法就是把部分工作交给其他人做。你所在的机构应该有很多志愿者，如果你能找到10个位于全国各地的志愿者，你可以委任他们为各地的区域协调人，这样他们每个人只要筹集100 000美元就够了。

筹集100 000美元比筹集1 000 000美元简单得多，但也绝非易事。这些区域协调人又怎么解决这个问题呢？他们也可以运用相同的策略把部分筹募工作交给别人。如果他们每个人都能找到另外10个筹募志愿者，那么这些筹募志愿者每人就只需筹集10 000美元即可。这种代理的过程可以层层深入下去直到筹款人可以一次募集到所有他们需要的捐款。因为平均每笔捐款额为100美元，所以志愿者完全可能找到一个人愿意捐献这么多善款，从而无需再找更多人来代理筹款的工作了。

可以将上述筹资策略用伪代码来表示，它的结构如下所示：

```
Void CollectContributions(int n)
{
    If (n<=100){
        从一个捐赠人处收集资金
    } else {
        找十个志愿者
        让每个志愿者收集n/10美元
        把所有志愿者收集的资金相加
    }
}
```

以上伪代码中，最重要的是

```
让每个志愿者收集n/10美元
```

这一行。这个问题就是原问题的再现，只是规模较原问题小一些。这两个任务的基本特征都是一样的（募捐n美元），只是n值的大小不同。再者，由于要解决的问题实质上是一样的，你 603
可以通过调用原函数来解决它。因此，前述的伪代码可以被下列代码取代：

```
CollectContributions(n/10);
```

需要着重指出的是，如果捐款数额大于100美元，函数CollectContributions最后会调用自己。在一段程序中，函数调用自身就是递归的特征。

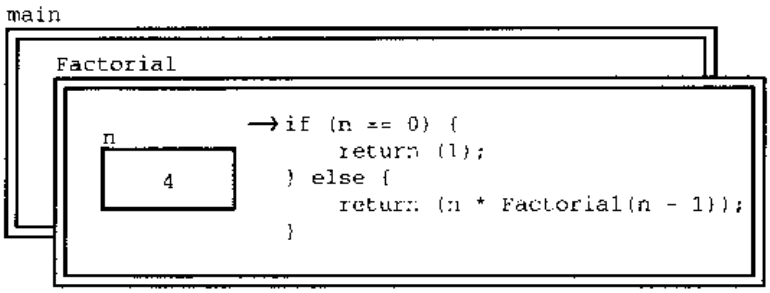
17.1.2 Factorial函数

CollectContributions这个例子之所以有用是因为它以一种浅显易懂的方式表达了递归的概念。但是，它没能告诉我们递归在实际中的用法，因为它用到的操作（如找10个志愿者，募集捐款等）都不能用C语言程序表达出来。要想从实践中了解递归的本质，我们需要找一些更适合程序设计的例子。

就像我们在第5章介绍阶乘函数时那样，用简单的数学函数来描述递归是最容易的。一个

```
fact = Factorial(4);
```

当main函数调用Factorial时，计算机创建了一个新的帧，并把实际参数值赋给形式参数n。Factorial的帧暂时取代了main的帧，如下图所示：



605

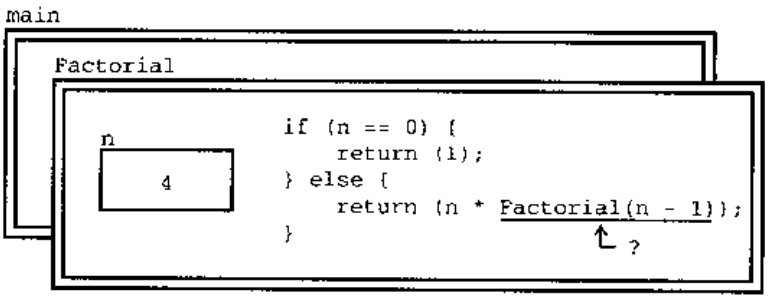
在上图中，Factorial函数体的代码显示在该帧中，这样可以使程序执行的当前位置更加清楚，我们用一个箭头表示这一点。在上图中，箭头出现在代码开始处，因为任何函数调用都是从函数体的第一句开始执行的。

现在计算机继续执行函数体，该函数从if语句开始。因为n不等于0，所以控制转而执行else子句，这时程序必须计算并返回表达式

```
n * Factorial(n-1)
```

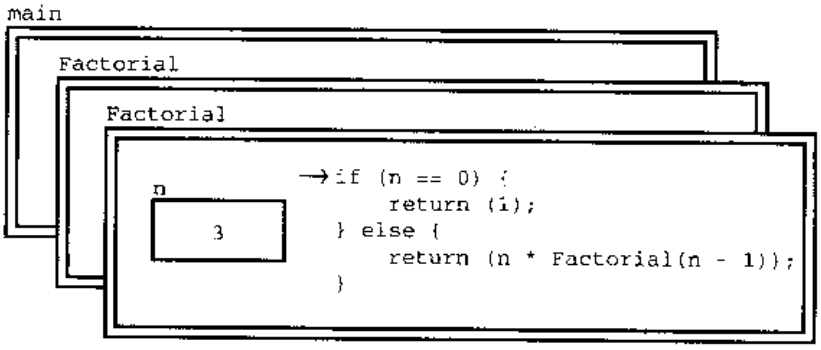
的值。

执行这个表达式需要计算Factorial(n-1)的值，这需要进行递归调用，当这个调用返回时，程序所需要做的只是将返回值乘以n。因此计算的当前状态如下图所示：



当函数Factorial(n-1)调用返回时，该返回值就替代图中下划线部分的表达式，使计算得以继续进行。

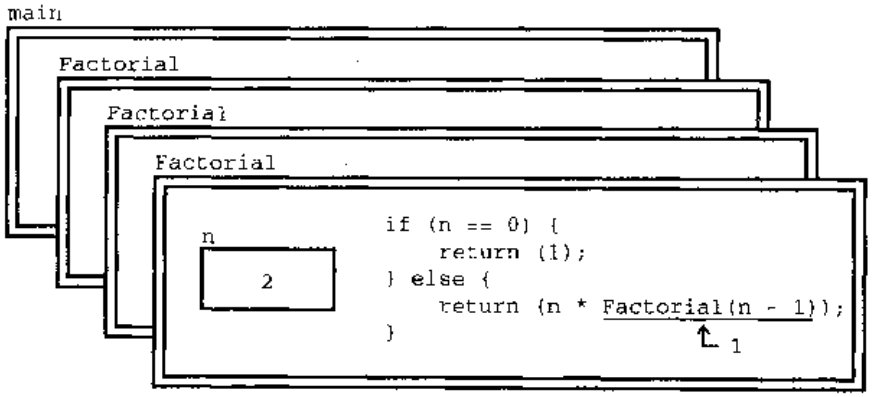
因此，下一步计算就是计算函数Factorial(n-1)的调用，这一步又是从计算实际参数表达式开始的。由于当前n值为4，所以实际参数表达式n-1的值为3。然后计算机再为Factorial创建一个新的帧，并把n-1的值赋给Factorial的形式参数。因此下一个帧如下图所示：



从这一点开始，计算就由一次次返回的递归调用所组成，用每一层Factorial的返回值计算下一层的值。例如，在现在这个帧中，函数Factorial(n-1)调用可以用1代替。因此这一层的结果可以表示为

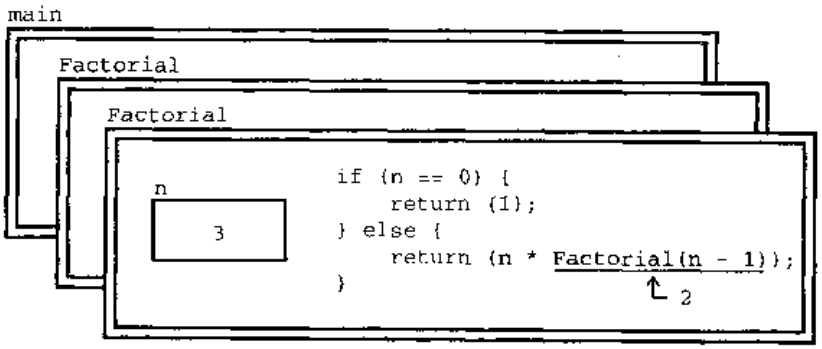
```
return (n * 1));
```

在这个帧栈中，n的值为1，所以这个调用的返回值为1，该值返回到其主调函数中，该函数可以用下图中最上层的帧表示：

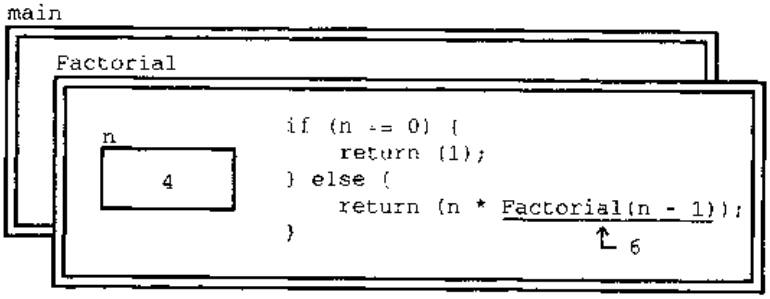


因为n现在为2，计算return语句会把2传递给前一层，如下图所示：

608



现在，程序把3×2返回到前一层，结果第一个被调用的Factorial函数的帧看上去为：



计算过程的最后一步包括计算4×6并把24返回给主程序。

17.1.3 递归信任

上一节中长长的Factorial(4)的例子告诉大家，计算机处理递归函数就像处理所有其他函数一样。当遇到一个递归函数时，至少在理论上可以模拟计算机的操作，并指出计算机要做的事。通过画出所有的帧并跟踪所有变量的值，可以重现计算机的整个操作过程并得出正确答案。然而，这样做往往会让人觉得整个过程更加复杂、更难理解。

当一个递归分解直接来自一个数学定义时，如在Factorial函数和RaiseIntToPower函数中那样，应用递归方法并不是那么困难。但是当问题本身变得越来越复杂时，情况就发生了变化，递归分解方法需要仔细思考才能找出来。下一节将解决一个更加复杂的问题，其中找到递归分解方法就比较困难。

17.1.5 排列的生成

许多文字游戏中的大部分工作是重新排列一组字母形成一个单词。例如，如果你想写一个Scrabble程序，最好能有一个工具把一组牌的所有可能的排列顺序都找出来。在文字游戏中，这样的排列通常称为**回文构词**（anagram）；在数学里，我们称之为**排列**（permutation）。

假设要写一个函数ListPermutations(s)，显示字符串s的所有排列。例如，调用

```
ListPermutations('ABC')
```

程序应该在屏幕上显示“ABC”的六种排列，如下所示：

```
ABC
ACB
BAC
BCA
CBA
CAB
```

六个排列的显示顺序并不重要，重要的是每种排列只能出现一次。

怎样编写这个ListPermutations(s)函数呢？如果只想使用循环控制结构，如for或while循环来解决问题，那么要想找到一个通用的解决方案来处理任何长度的字符串的排列问题就比较困难了。但是如果用递归思想来考虑这个问题，我们就能得到一个相对简单的解决方案。

和一般的递归程序一样，解题过程的难点在于怎样把复杂的原问题分解成简单的同类问题。在这个例子中，要为一个字符串找出所有可能的排列，你需要搞清楚为一个稍短一些的字符串生成所有排列对解题有没有帮助以及有何帮助。 611

为了对这个问题有更清楚的认识，我们来探讨一个具体的例子。假设你要为一个含5个字母的字符串“ABCDE”生成所有可能的排列。在解决方案中，你可以应用递归信任生成任何稍短的字符串的所有排列。假设递归调用是有效的，并且都已经成功返回结果了。现在，这个关键性的问题又出现了，也就是怎样利用稍短字符串的排列解决5字母字符串的排列问题。

解决这个排列问题的关键是了解以下事实：5字母字符串“ABCDE”的所有排列是由下列字符串组成的：

- 字母"A"后跟着"BCDE"的所有可能排列。
- 字母"B"后跟着"ACDE"的所有可能排列。
- 字母"C"后跟着"ABDE"的所有可能排列。
- 字母"D"后跟着"ABCE"的所有可能排列。
- 字母"E"后跟着"ABCD"的所有可能排列。

更一般化地说，要显示长度为n的字符串的所有排列，可以依次把n个字母中的每个字母置于列首，其后跟着剩下$n-1$个字母的所有排列。

这种解决方法的唯一问题在于递归子问题和原问题的形式并不完全相同，而这正违反了

的。函数PermuteWithFixedPrefix完成了所有必需的工作，但它并不是所需要的原型。当你调用PermuteWithFixedPrefix时，你需要传递一个整数和一个字符串。大多数客户只考虑排列一个字符串中的字符，这里面不包括任何整数。因此，从客户的角度考虑的话，应该像下面这样定义ListPermutations：

```
static void ListPermutations(string str)
{
    PermuteWithFixedPrefix(str, 0);
}
```

613

当用递归方案解决问题时，往往需要定义一个原型与原问题稍有不同的递归函数。就像在PermuteWithFixedPrefix中一样，新的函数经常带有一些额外的参数，但这些参数对于整个问题来说并不是必须的。这些参数记录了递归算法的进程，并且为衡量子问题是否更简单提供了一个标准。客户所用的函数调用了内部函数，给这些额外参数传递初值，就像在ListPermutations中那样。像ListPermutations那样的函数，它的目的是为一个更通用的函数提供额外参数，那么这样的函数叫做**包装**（wrapper）。

17.1.6 用递归的思想思考

学习使用递归比学习任何其他程序设计知识更要求你从整体上考虑程序设计过程。如果采用归约的方法，跟踪一个复杂递归分解的所有步骤，你最终将不得不面对大量的细节问题，而这些细节最好留给计算机去面对。前面曾介绍的Factorial(4)的整个计算过程的步骤演绎出来要用好几页纸。相比之下，整体的思想

```
Factorial (4) = 4 * Factorial (3)
```

只需要一行就可以表达清楚了。这两种递归思路在概念复杂度方面有巨大差别。从整体的观点看问题，可以降低大多数递归程序的复杂度，使你能够理解这种解题方法。然而，如果你因为对解决方案的正确性心存疑虑而选用归约的方法，大量的细节将不可避免阻碍你理解这种解决方案的复杂结构。

以正确的方法思考递归问题并非易事。要学会有效地应用递归，就需要不断地实践。对于很多学生来说，掌握这种概念需要几年的时间，但因为递归是最有用的程序设计技能之一，所以下这些功夫也是值得的。

17.2 抽象数据类型

本书介绍过，程序中的数据结构形成了一个层次结构。原子数据类型（如int、char、double和枚举类型）构成了层次中的最低层。为了表达更复杂的信息，必须把原子类型组合起来构成大一些的数据结构。而这大一些的结构又可以组合成为更大的数据结构，而且这种结合是开放式的，没有限制的。使用三种构造类型的基本元素（数组，指针，记录），可以在这个层次结构中加入新的层次。如果有一种已存在的类型，你可以定义这个类型的数组，或者把它当作一个记录的一个字段，或者声明一个指向它的指针。这三种工具是数据层次结构的粘合剂，你可以用它们建造具有任意复杂度的结构。

614

然而，当你学到更多关于程序设计的知识和技能后，你会发现特殊的数据结构有时非常有用，并且值得学习。例如，一个字符串在计算机内部表示为一个字符数组，而数组又是由一个指向数组首字母的指针来表示。在本书中，第14章介绍了字符串的基本结构。但字符串也有它的抽象行为，这些抽象行为比它的表示更重要。在第9章已经介绍过了它的抽象属性。

信放进一个队列，稍后再处理它。同样，大多数支持多用户的系统都把文件排队进行打印。当你想打印一份文档时，那个文档先进入系统的某个队列中，当打印机完成之前的那些打印请求后，就开始打印你的文档。队列的操作是一样的，但队列中存储的信息则完全不同。

关键在于使用队列软件包的客户必须决定存储在等候队列中的数据的类型。如果客户是一个电子邮件系统，那么它需要将邮件排队。如果客户是一个打印文件的程序，那么它需要排队打印请求。如果客户要的是一个模拟超市的程序，那么它必须能将以某种方法表示的超市顾客进行排队。而就队列软件包而言，它的职责是在客户提供一个数据时，它该怎样进行入队操作。当稍后执行一个出队操作时，队列软件包必须能把这个数据完整精确地送回给客户。

因此，为了使你的队列软件包尽可能通用，就应该让你的客户来控制存储在队列中的数 616
据类型。唯一的问题是你必须在接口中引用这个类型。和其他程序设计语言不同，C语言中没有一个能与任何类型相匹配的特殊类型。在C语言中，最好是用void*这个类型，它能匹配任何指针类型。如果使用void*，那么队列软件包的客户就能对任何指针类型的数据进行出队、入队操作了。因为我们总是可以为非指针型数据分配空间并使用其地址，所以在实践中，这种限制并不是什么大问题。

下一问题是怎样表示一个等候队列本身。与顾客不同，队列是实现的一个属性。队列软件包要负责在队尾使新数据入队，在队首使头一个数据出队。你的实现必须能够控制等候队列的表示，使它能进行这些操作。因此，在这个接口中，你要定义抽象队列类型queueADT，让实现部分控制基本的表示。

在C语言中，你可以在接口中定义一个类型，使它的内部表示对客户来说不可见。要做到这一点，你必须在接口中包含一个抽象类型的定义，其形式如下面的语法框所示。例如，为了把queueADT定义为一个抽象类型，接口必须包含下列定义：

```
typedef struct queueCDT *queueADT;
```

C语言的语法认为，这一行把queueADT类型定义为指向名为queueCDT的结构的指针。而queueCDT还没有被定义。因为指针总是有同样的大小，所以C编译器允许你使用结构指针，即使它还不知道结构的任何细节。一个没有被定义的结构称做**不完全类型**（incomplete type）。

语法：抽象类型的定义

```
typedef struct nameCDT * nameADT;
```

其中：

*name*ADT是抽象类型的名字。

*name*CDT是对应的具体类型的名字。

在实现中，通过写这个结构的定义来完成该类型，如下面的句法框所示。该定义包括用于标识该不完全结构的名称，它被称作**结构标记**（structure tag）。类型*name*CDT是与接口中定义的抽象类型*name*ADT相关的具体数据类型。接口的客户只能看到抽象类型。客户可以声明抽象类型的变量，对这些变量的值进行操作，就好象它们是原子类型那样。因为编译器并不知道它的基本表示是怎样的，所以客户也不能间接引用该抽象类型去看它的具体构造。另一 617
方面，这个实现也已经访问了具体类型。因为编译器现在知道了基本结构，所以实现中的函数可以间接引用抽象类型指针，从而看到具体类型中的字段。

```
/*
 * Function: FreeQueue
 * Usage: FreeQueue(queue);
 * ------------------------
 * This function frees the storage associated with queue.
 */

void FreeQueue(queueADT queue);

/*
 * Function: Enqueue
 * Usage: Enqueue(queue, obj);
 * ---------------------------
 * This function adds obj to the end of the queue.
 */

void Enqueue(queueADT queue, void *obj);

/*
 * Function: Dequeue
 * Usage: obj = Dequeue(queue);
 * ----------------------------
 * This function removes the data value at the head of the queue
 * and returns it to the client.  Dequeueing an empty queue is
 * an error.
 */

void *Dequeue(queueADT queue);

/*
 * Function: QueueLength
 * Usage: n = QueueLength(queue);
 * ------------------------------
 * This function returns the number of elements in the queue.
 */

int QueueLength(queueADT queue);

#endif
```

图 17-1 (续)

619

17.2.4 实现队列抽象

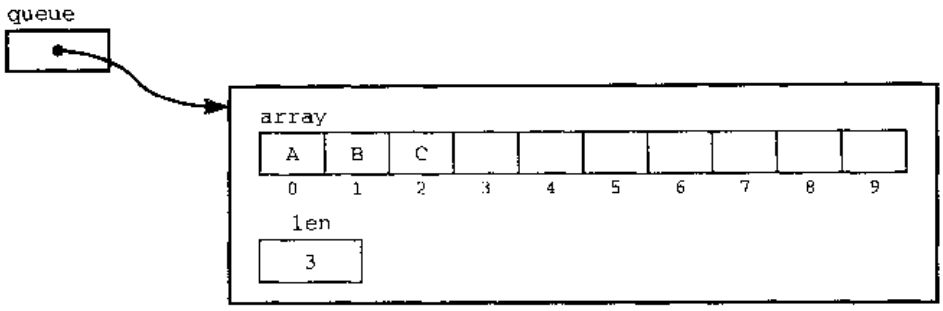
一旦完成了接口的设计，下一步就是实现队列软件包。作为实现的一部分，需要定义等候队列本身的基本表示方式。最直接的表示是一个数组和一个指示其有效长度的整数。因此，具体的类型定义如下所示：

```
struct queueCDT {
    void *array[MaxQueueSize];
    int len;
};
```

其中MaxQueueSize是一个常量，它指明了在一个队列中最多能保存的元素个数。

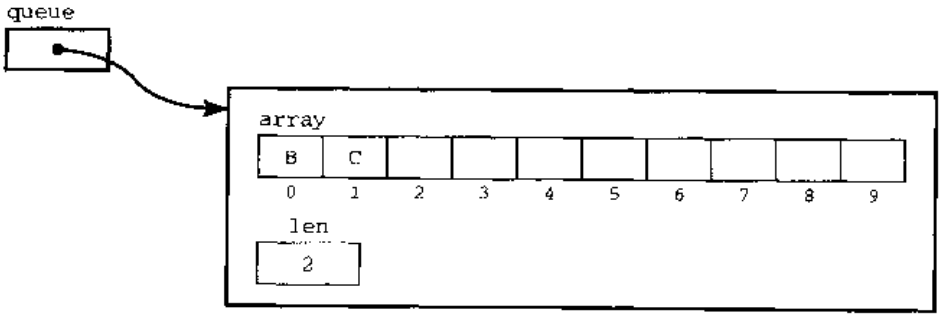
要实现函数NewQueue，要做的是为具体的结构分配空间并将队列长度初始化为0。它的实现代码如下：

```
queueADT NewQueue(void)
```



621

那Dequeue操作又如何呢？在上图所示的队列中，Dequeue操作会将A从队列中删去，并把它作为调用Dequeue的返回值返回。给定了当前的表示，唯一能把第一个元素从队列中删去的方法就是把所有其他元素都左移一位，最终的队列如下所示：



使用这种方法，Dequeue的实现如下所示：

```
void *Dequeue(queueADT queue)
{
    void *result;
    int i;

    if (queue->len == 0) Error("Dequeue of empty queue");
    result = queue->array[0];
    for (i = 1; i < queue->len; i++) {
        queue->array[i - 1] = queue->array[i];
    }
    queue->len--;
    return (result);
}
```

queue.h接口的完整实现在文件queue.c中给出，如图17-2所示。

```
/*
 * File: queue.c
 * -------------
 * This file implements the queue.h abstraction using an array.
 */

#include <stdio.h>

#include "genlib.h"
#include "queue.h"

/*
 * Constants:
```

图17-2 queue.c

```
    queue->array[queue->len++] = obj;
}

/*
 * Function: Dequeue
 * -----------------
 * This function removes and returns the data value at the
 * head of the queue.
 */

void *Dequeue(queueADT queue)
{
    void *result;
    int i;

    if (queue->len == 0) Error("Dequeue of empty queue");
    result = queue->array[0];
    for (i = 1; i < queue->len; i++) {
        queue->array[i - 1] = queue->array[i];
    }
    queue->len--;
    return (result);
}

/*
 * Function: QueueLength
 * ---------------------
 * This function returns the number of elements in the queue.
 */

int QueueLength(queueADT queue)
{
    return (queue->len);
}
```

图 17-2（续）

17.2.5 队列抽象的另一种实现方法

上一节开发的队列软件包的实现不是唯一的选择。尽管它很简单，但它有以下两个局限性：

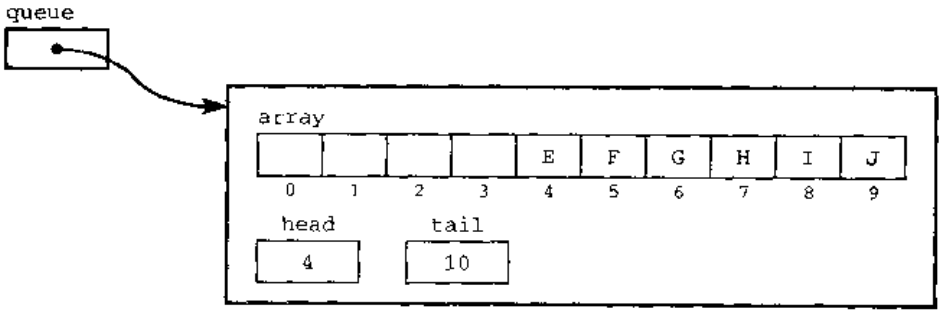
1) 调用Dequeue效率不高。每次调用Dequeue后，该实现需要把队列中剩下的项都向数组头的方向移动。对于有很多项的队列来说，这是很费时的。如果客户要把队列软件包应用于有时间要求的工作中，Dequeue的低效率就变得不可接受了。

2) 队列的最大长度是固定的，由常量MaxQueueSize指定。有些客户可能无法接受这种限制。对它们来说，只要内存允许，队列最好能动态扩展。

如果为队列结构选择一个不同的基本表示，可以同时解决这两个问题。另外，因为接口将queueADT定义为抽象类型，你可以任意改变队列的基本表示而不影响接口的客户。

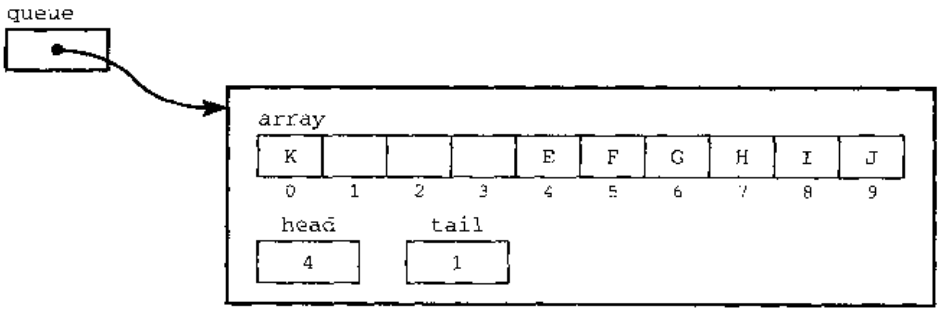
第一步，考虑怎样解决Dequeue的低效率问题。低效的根源在于函数必须对数组中的每一个元素进行移位操作，以保证第一个队列项处于下标位置0处。尽管这样简化了队列软件包的实现，但它并非是必须的做法。可以用跟踪两个下标位置的方法解决：等候队列中的第一项的下标和下一个入队的项将存入的位置的下标值。程序员称这两个位置为**队头**（head）和**队尾**（tail）。包含这两个下标的具体的类型定义如下：

G、H、I和J还在队列中。其状态如下图所示：

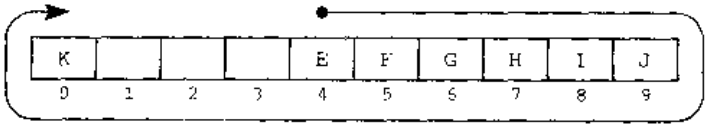


626

如果这时客户K来了会怎样呢？数组末尾没有多余的位置了。实际上，tail的下标值现在指示的元素已超过数组范围。另一方面，数组却并非处于“满”的状态。因为前四个位置空出来了。如果小心使用下标的话，你在实现队列软件包时，可以使Enqueue操作重新使用Dequeue操作释放的空间。在当前的例子中，K可以进入数组元素0处，使队列处于如下状态：

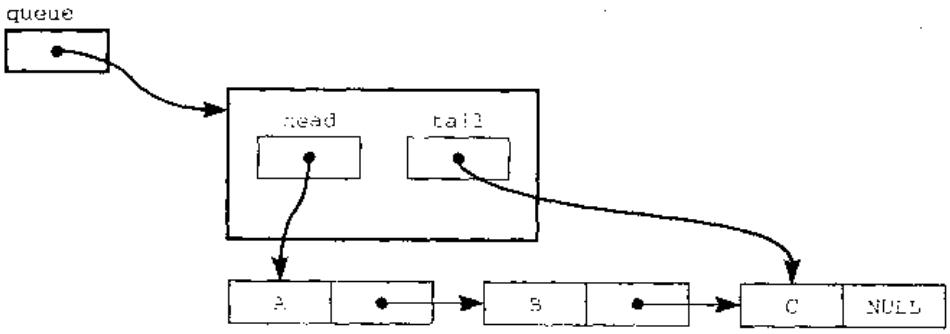


在这个设计中，数组末尾“卷回”到开头，这样数组更像一个环而非一个线性表了，如下图中的箭头所示：



因为数组开头与末尾在概念上是互相连接起来的，所以程序员称这种表示为**环缓冲区**（ring buffer）。环缓冲区常常用于实现队列，在练习7里读者有机会实现这种设计。

如果你继续深入学习计算机科学，你会发现队列常常用一种完全不同的方法实现。除了用数组保存队列外，还可以设计一种队列表示方法，其中每个队列项包含一个指向下一个队列项的指针。例如，一个队列包含A、B、C三项，如下图所示：



627

因为这个队列的每个项包含一个指向下一个项的指针，所以这种结构被称为**链表**（linked

如，考虑一下图17-2中的队列软件包的原始实现。Enqueue操作的运行时间不随问题规模的变化而改变，此处问题规模可以定义为队列中当前的项数。在计算机科学中，如果一个操作所需的时间与问题规模无关，则说该操作以**恒定时间**（constant time）运行。在*O*标记中，恒定时间用*O*(1) 表示。*O*(1) 的意思是当*N*变大时，这个程序的运行时间随 1 的改变而改变。因为1是常数，所以当*N*值增加时它并不发生变化，这也是恒定时间操作的一个显著特征。

图17-2的Dequeue操作的行为有所不同。Dequeue的实现用下列for循环把该队列中每一项向数组头方向移一步：

```
for (i = 1; i < queue->len; i++) {
    queue->array[i - 1] = queue->array[i];
}
```

如果队列包含*N*项，这个for循环就要执行*N*个周期。随着*N*的增加，for循环执行时间成正比例增加。而且，如果*N*很大，for循环的消耗远远超过循环外所有操作的消耗，因为无论*N*值多大，这些循环外的操作往往只执行一次。这样一来，在某一范围内随着*N*变大，Dequeue 629
操作的运行总时间会随*N*的增加而增加。如果一个操作的运行时间与问题规模成正比，则称此操作以**线性时间**（linear time）运行，用*O*标记表示为*O*(*N*)。

在本章前面的17.2.5节中提到的队列软件包的环缓冲区实现的优点在于，它把Dequeue操作的运行时间由线性时间降低为恒定时间，即从*O*(*N*) 到*O*(1)。如果*N*值很大，这种优势是很明显的。另一方面，使一个线性算法以恒定时间运行所节省的时间远远小于在更复杂的算法中进行改进带来的省时效果。本章剩下的内容从一个新的视角看看第12章提出的排序问题，以强调计算复杂性的问题。

17.3.3 再看选择排序

在第12章中，用于实现SortIntegerArray的算法称为*选择排序*。如果给出一个有*N*个元素的数组，选择排序算法要访问每一个数组元素的位置，并确定该位置应该存放的值。为了找到合适的值，算法必须查找剩下的数组元素并找到最小的值。这样，算法用*N*步填好第一个位置，用*N*－1步填入第二个位置，依次类推，因此，总的运行时间与

$$N+N-1+N-2+\cdots+3+2+1$$

成正比。第12章的12.2.5节介绍过，该公式可化简为：

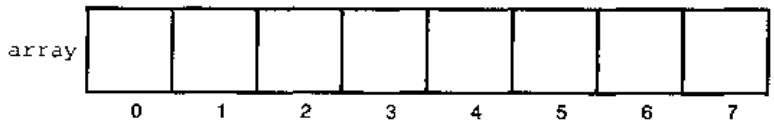
$$\frac{N^2+N}{2}$$

当使用*O*标记来估计一个算法的计算复杂性时，目的是提供一种量化手段，以便衡量当*N*变大时，*N*的变化是如何影响算法的性能。因为*O*标记并非一种精确的量化手段，所以我们最好能简化括号中的表达式，以便用最简单的形式量化算法的行为。

对圆括号里的公式应用下列步骤，可以简化*O*标记。

1) *消去公式中任何随N变大而变得无足轻重的项*。例如，在选择排序算法中，当*N*值增加时，N^2很快比*N*大很多。选择排序的运行时间因此更依赖于N^2项。因此，当使用*O*标记时，可以忽略*N*而重点关注N^2。

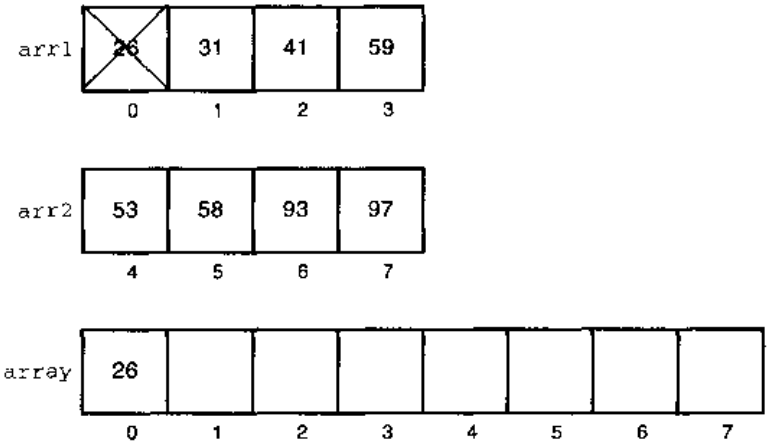
2) *消去任何常数系数*。当你计算计算复杂性时，你最关心的是对于不同的*N*值，各种算法 630
的相对运行时间。如果你用一个比率表示相对时间，那么分子和分母中的常数系数都会被消去。常数系数对于相对运行时间是没有影响的，所以可以在用*O*标记时删去常数系数。



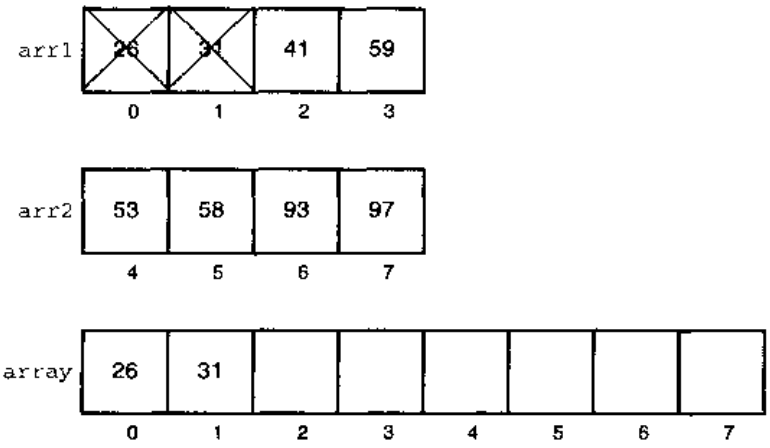
你该怎么做呢？

17.3.5 合并两个数组

把已排序的子数组重组成一个完整数组比排序本身简单多了。这种技术叫**合并**(merging)。它基于一个事实，即完整的排序的第一个元素只能是arr1或arr2的第一个元素，但要取更小的那个元素作为第一个元素。在这个例子中，新数组中的第一个元素是arr1中的26。如果 632
把这个元素放入array[0]，实际上，就是把它从arr1中划掉，结果得到以下结构：



接着，下一个元素只能是两个子数组中第一个未用过的元素。把arr1中的31和arr2中的53比较，并选择前者：



可以继续在arr1、arr2中选择较小值的过程，直到整个数组被填满为止。

17.3.6 合并排序算法

合并操作与递归分解相结合，产生了一个新的排序算法——**合并排序**(merge sort)，实现这个算法的方法很简单。函数SortIntegerArray首先判定数组的大小。如果数组没有元 633
素或只有一个元素，那么数组必然已经被排过序。因此这个条件定义了这种简单情况。然而，如果数组包含的元素多于1个，就需要执行下列步骤。

1) 把数组分成两个较小的子数组，每一个数组的大小是原数组的一半。

当调用合并排序的实现（SortIntegerArray函数）为N个数字排序时，运行时间可分成下面两部分：

1) 在当前的递归分解层次上，执行操作所需的所有时间。

2) 执行递归调用的时间。

在递归分解的最上面一层，执行非递归操作所消耗的时间与N成正比。SortIntegerArray的两个for循环一起形成N个循环周期，对Merge的调用填满了原数组的N个位置。如果你把这些操作加起来，去掉常量因子，你会发现任何一次SortIntegerArray调用的复杂性（如果不考虑其中的递归调用）需要$O(N)$次操作。

但递归操作的代价又如何呢？要排序一个大小为N的数组，必须递归地排序两个大小为$N/2$的数组。为每一个子数组排序都需要同样的时间。如果应用同样的逻辑，则很快就能够确定每一个递归调用需要的时间正比于该层上的$N/2$，加上递归调用所需时间。继续同样的过程，直到你到达一种简单情况，即子数组只有一个元素或无元素为止。

解决这个问题所需的总时间是递归分解的每一层所消耗的时间之和。总的来说，分解的结构如图17-3所示。当你沿着递归层次向下执行时，数组越来越小，但数组个数却越来越多。然而，每一层的工作量总是和N直接成正比的。因此，确定工作总量就转化为确定层数的问题。

在递归层次结构的每一层中，N值都除以2。总层数等于你在$N=1$之前将N除以2的次数。 635
用数学术语表述这个问题，就是说必须找到一个这样的k值，使得

$$N=2^k$$

解这个方程，得到k的值为：

$$k=\log_2 N$$

因为层数为$\log_2 N$，且每一层的工作量与N成正比，所以总工作量与$N\log_2 N$成正比。

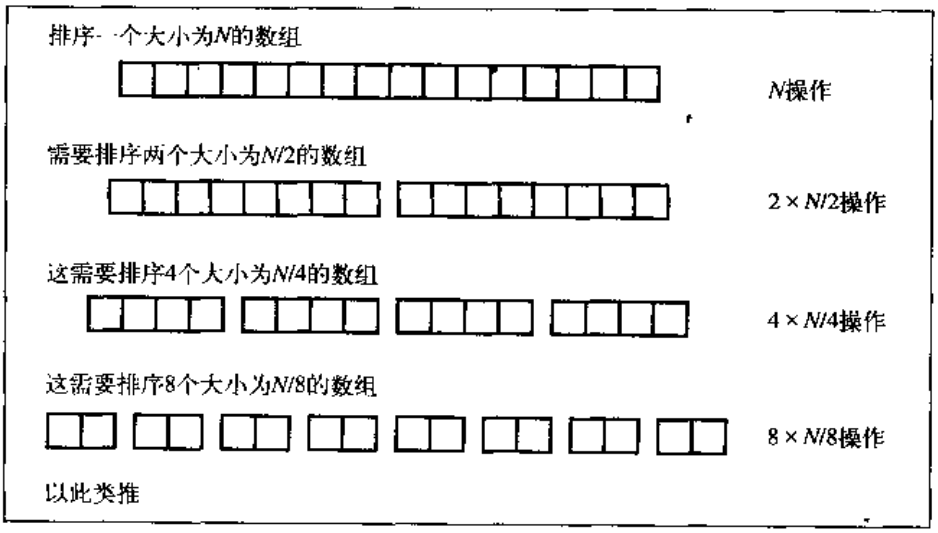


图17-3 合并排序的递归分解

在其他学科中，对数的底数通常为10（常用对数）或数学常量e（自然对数）；但在计算机科学中，总是使用**二进制对数**（binary logarithm），即以2为底数的对数。由于对数计算底数不同，而底数又是常数，因此，当我们谈到计算复杂性的问题时，我们可以像以前那样忽略对数的底数。这样合并排序的计算复杂性可以写为：

$$O(N\log N)$$

的变化而变化的。

• 降低一个算法的计算复杂性可以大大提高运行效率。

复习题

1. 定义术语*递归*。

2. 递归与逐步精化的本质区别是什么？

3. 术语*递归信任*是什么意思？

4. 典型的递归函数的函数体中第一个关键字是什么？

5. 在PermuteWithFixedPrefix的实现中，两次调用了ExchangeCharacters。第二次调用是必须的吗？说明理由。

6. 什么是包装？为什么包装在递归程序设计中如此普遍？ 638

7. 队列的基本操作是什么？

8. 假设你想设计一个抽象类型PictureADT来保存图形结构。在不知道任何基本表示的情况下，怎样在接口中定义PictureADT，以使它能和相应的具体类型相联系？

9. 一个具体类型的定义属于接口还是实现？

10. 什么是环缓冲区？将环缓冲区作为队列的基本表示主要有哪些优点？

11. 为了简化O标记的括号中的公式，可以做哪些修改？

12. 以O标记表示的选择排序的计算复杂性是什么？合并排序的复杂性又是什么？描述其区别说明了什么问题。

13. 图12-3中实现的二分查找算法的计算复杂性是什么？

程序设计练习

1. 在第4章的练习8所介绍的Fibonacci数列中，前2项为0和1，以后的每一项是该项的前2项之和。因此这个数列的开始几项为：

$$
\begin{array}{llll}
F_0 & = & 0 & \\
F_1 & = & 1 & \\
F_2 & = & 1 & (F_0 + F_1) \\
F_3 & = & 2 & (F_1 + F_2) \\
F_4 & = & 3 & (F_2 + F_3) \\
F_5 & = & 5 & (F_3 + F_4) \\
F_6 & = & 8 & (F_4 + F_5)
\end{array}
$$

用同样的方式生成后续项。写出函数Fib(n)的递归实现，使它返回第n个Fibonacci数。你的实现必须只依赖于数列中项之间的关系，而不能用任何迭代结构，如for或while。

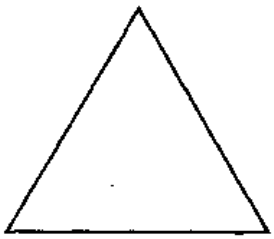
2. 第5章介绍的数学中的组合函数C(n, k) 常常用阶乘来定义，其定义如下：

$$C(n,k)=\frac{n!}{k!\times(n-k)!}$$ 639

C(n, k) 的值也可以排列成一个三角形，其中n值由上至下增大，k值从左向右增大。这个结构叫Pascal三角形，如右图所示。第5章的练习12指出，Pascal三角形有一个有趣的特征，除了左、右两边的项都为1以外，它的每一项都为其上两项之和。例如，考虑下页的图所示的Pascal三角形中被圆圈住的项：

C(0, 0)
C(1, 0) C(1, 1)
C(2, 0) C(2, 1) C(2, 2)
C(3, 0) C(3, 1) C(3, 2) C(3, 3)
C(4, 0) C(4, 1) C(4, 2) C(4, 3) C(4, 4)

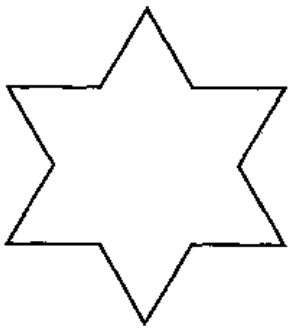
最早的一种分形之一是*koch*雪花，这是以它的发明者Helge von koch命名。Koch雪花以一个等边三角形开始：



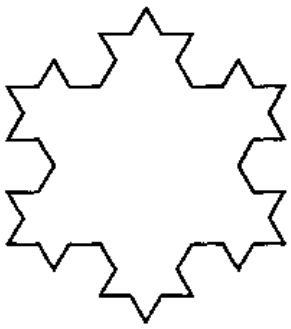
它被称为0阶koch分形。之后这个图形被逐步修改生成了更高阶的分形。在每一步修改中，三角形的每一条边被分为三段，中间一段线段被一个三角形的凸形取代。这样，如果你把三角形的每一条边都用下图做改变：



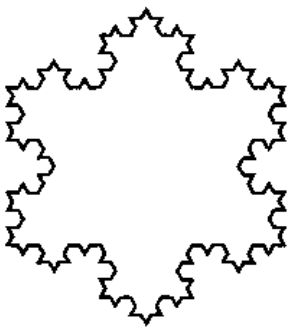
你将得到1阶koch分形：



如果你把这个图形中的每一条边再用一条有凸起三角形的线段取代，你将得到2阶koch分形： 642



再次用同样的方法替换图中的每一条边，将得到下图所示的3阶koch分形，看起来像一片雪花：



9. 在第16章的练习8中，要求你实现一个字典模块，能够存储键值和它们的相关定义。然而，按照该练习给定的设计思路，整个程序中只能有一个字典。在某些应用中，我们可能需要多个字典用于不同的目的。

重新设计dict.h接口，以实现下列变化：

a. 定义一个抽象数据类型dictionaryADT，使它能够存储所有用来表示一个字典的信息。

b. 用函数NewDictionary替代InitDictionary函数，NewDictionary能分配新空间，并返回一个新的dictionaryADT。

c. 输出函数FreeDictionary，它能释放一个已存在的字典的空间。

d. 在Define和Lookup的原型中，增加一个类型为dictionaryADT的参数，使它们知道目前正在使用的是哪个字典。

10. 如表17-2所示，选择排序对于小数组来说比合并排序快，尽管当数组变大时，情况相反。如果你想设计一个函数SortIntegerArray，使之适应于各种大小的数组，那么可以把两种策略结合起来，对大数组用合并排序，对小数组使用选择排序。用这种方法重新实现函数SortIntegerArray，那么你该如何选择区分大小数组的那个分界点呢？ 645

附　录

附录A　C语言的语法和结构总结

本附录提供了本书中出现的关于ANSI C的语法规则和结构的快速参考。尽管本附录确实包括了大多数重要的特性，但并不是C语言的参考指南。要找到关于某个特殊主题的更多信息，请参考相应的章节。

A.1　编译过程概述

一个C语言程序由一个或多个源文件组成，每一个源文件又由可读的文本组成，这些文本是用称为ANSI C的高级程序设计语言写成的。C编译器把每一个源文件翻译成相应的目标文件，目标文件中包含着在一个特定的计算机系统上执行程序所需要的指令。然后，将这些目标文件以及其他用于实现各种常用操作的称为库的目标文件链接起来创建一个可执行程序。

编译一个源文件的过程包括下列四个阶段：

1) *预处理*。在预处理时，C编译器用一组已有的操作变换整个源文件的文本，这些操作包括读入库接口和以符号常量的定义替代符号常量名。

2) *词法分析*。在词法分析阶段，编译器收集经过预处理的源文件中的字符以形成有意义的单元，这种单元叫*记号*。

3) *语法分析*。这个阶段包括检验源文件中的记号是否符合C语言的语法规则，或者句法规则。此外，语法分析阶段还负责解释程序的意义，即语义。

4) *生成代码*。编译的最后一个阶段包括把经过语法分析的程序表示翻译成执行程序所必须的机器指令。尽管这一阶段对生成目标文件是必须的，但它对源文件并没有直接影响，因此与程序员没什么关系。

前两个阶段，即预处理和词法分析，将在A.2节和A.3节中分别讨论。A.4～A.8节将着重讨论语法分析阶段并回顾C的语法结构。A.9节回顾本书用到的ANSI库函数。

A.2　C语言预处理器

预处理器完成C语言中的某些功能，本书仅详细讨论了其中的两个功能，即定义常量和包含接口文件。它们将在下面两个小节中讨论。

646～648

A.2.1　#define规范

形如

```
#define name definition
```

的源程序行定义符号*name*等价于*definition*。在源文件剩下的部分中，预处理器寻找任何作为一个完整的记号出现的*name*，用*definition*中的字符替换*name*。例如，在遇到语句

```
#define Pi 3.14159265
```

用于分隔源文件中的其他记号，否则就被忽略掉。因此空格可以帮助人们理解代码的结构。习惯上，对程序的每一行的缩排都是为了使控制结构更为清晰。同样的，空行用来分隔程序的不同部分。

A.3.2 注释

注释由字符/*和*/之间的文本组成。注释文本可以跨越很多行，但一个注释不能嵌套在另一个注释中。注释文本是为了方便程序的读者，但编译器会忽略注释。 650

A.3.3 标识符

程序中的变量、函数、类型等的名字被统称为标识符。在C语言中，构造标识符的规则为：

1) 标识符必须以字母或下划线开头。

2) 标识符中所有其他字符必须为字母、数字或下划线，不能有空格或其他特殊字符。

3) 标识符不能使用下列关键字：

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

标识符中的大写和小写字母是不同的。例如，名字ABC和名字abc是不同的。标识符可以为任意长度，但C编辑器在判断两个名字是否相同时不考虑第31个字符以后的字符。某些实现可能会对模块间共用的标识符附加一些额外的限制。

A.3.4 常量

程序可以包含下列通用类型的常量：整型、浮点型、字符型和字符串。这些类型将在下面讨论。每一种类型常量的构成规则如下：

- 整型。整型常量一般写成一个数字字符串，以表示一个十进制的数。如果这个数以0开头，则编辑器把这个数当作八进制整数，因此，常量040被当作一个八进制数，代表的十进制数为32。如果你在一个数字常量前加前缀0x，那么编译器将把它解释为十六进制数。因此，常量0xFF对应的十进制数为255。你可以在数字串末尾加字母L，明确表示该数字是一个长整型常量。因此常量0L等于0，但这个值是长整型的。同样，如果你把字母U作为后缀，这个常量被当作是无符号常量。
- 浮点型。C语言中的浮点常量有小数点。因此，如果程序中出现2.0，那么这个数在机器内部表示为一个浮点型的值；如果程序员写成2，那么这个值将是一个整数。浮点型常量也可以用科学计数法写成*x.xxxx*E±*yy*的形式，表示*x.xxxx*乘以10的*yy*次方。 651
- 字符型。在C语言中你可以用单引号把字符括起来表示字符型常量。该字符的值由编码模式（通常为ASCII）给出，该模式将每一个字符映射成一个数值。除了标准字符外，C语言还支持下列表示特殊字符的转义序列：
 - '\a' 震铃字符（终端的嘟嘟声）
 - '\b' 退格
 - '\f' 换页（开始一个新的页面）
 - '\n' 换行
 - '\r' 回车（返回至本行首位）

A.5 语句

本节回顾C语言中可用的语句形式。

A.5.1 简单语句

C语言中最常见的语句是简单语句，它由一个表达式后面跟一个分号组成：

```
表达式;
```

A.5.2 空语句

一个分号本身组成了一个没有任何作用的语句。这种语句形式叫做*空语句*，常用在一个控制结构体中，在这种控制结构中，重要工作都在头语句中完成了。

A.5.3 语句块

可以用花括号把几条语句括在一起，形成一个*语句块*：

```
{
    …可选的声明…
    …语句…
}
```

653

然后你可以将所产生的语句块作为任何控制语句中的一个语句。如范例所示，在任何块中的语句之前可以有变量声明。在本书中，变量声明只用于定义函数体的块内。声明将在本附录后面A.6节中深入讨论。

A.5.4 if语句

if语句用来表示简单的条件执行，并以下面两种形式出现

```
if (表达式) 语句
```

和

```
if (表达式) 语句1 else 语句2
```

A.5.5 switch语句

switch语句在一个表达式的值指定了几种相互独立的情况中的一种时使用。该语句的形式为：

```
switch (e) {
  case c1:
    语句
    break;
  case c2:
    语句
    break;
…更多的case子句…
  default:
    语句
    break;
}
```

可以有任意多个case子句，但每一个case子句所用的值必须是某种标量类型的常量。default子句是可选的，但推荐使用这个子句，如果没有default子句，并且switch行中的值不能与任何一个case子句的值相匹配，则程序会从switch语句退出，不执行任何动作。

出现在switch语句中break语句并非C语言的语法所必须的。当一个break语句被执行时，控制立刻转移到整个switch语句后面的语句。如果break语句（或其他任何改变控制

言程序设计所必须的。更重要的是，这些控制形式很容易因为滥用而造成程序结构的复杂化。下面将分别描述这三种语句，这样你在实际中遇上它们时就能理解其作用了。

do语句类似于while，只是测试是在循环末尾而非开头进行。do语句的一般形式为：

```
do{
    语句
}while (条件表达式);
```

尽管人们通常更喜欢用while，但这个语句在有些场合很有用。而用do语句的程序更容易出错，因为do循环体至少运行一次，哪怕条件表达式一开始就为FALSE也是这样。

continue语句的形式为：

```
continue;
```

它使程序立即开始执行离循环最近的下一个循环周期。尽管这非常方便，但它也使程序的可读性降低，尤其是用于复杂循环结构时更是如此。

C语言可用的最低级的控制是goto语句，其一般形式为：

```
goto label;
```

标识符*label*指定当前函数中的某一条语句，该语句用标识符*label*及一个冒号作为标记，如下所示：

```
label: 语句
```

goto语句使控制直接转到被*label*标记的语句。而goto的不规则的使用会破坏程序的结构，使它变得极其难懂，难以维护。本书中没有使用goto语句，建议你也不要使用它。 656

A.6 函数

C程序中的所有可执行语句都出现在一个函数体内。当计算机执行一个程序时，它从调用main函数开始。而main函数又调用其他函数，这些函数中分别解决一部分问题；这些函数可能又调用别的可以进一步细化问题的函数。把一个问题分解成更简单的部分的过程叫做逐步精化。

A.6.1 函数定义

函数定义有如下的句法形式：

```
result-type  name(argument-specifiers)
{
    ...optional declarations...
    ...body...
}
```

在这个范例中，*result-type*是函数返回值的类型，*name*是函数名，*argument-specifiers*是一个形式参数声明表，形式参数之间以逗号隔开。*optional declarations*部分包含函数中使用的局部变量的声明。而*body*则由实现函数所需的语句组成。

A.6.2 函数原型

在C程序中使用某一函数前，应该先通过指定它的原型[⊖]来声明它。原型和函数定义的形式完全相同，只是原型的后面不是整个函数体而是一个分号。在原型中可以不写形参变量名，尽管提供参数名通常对读者是有帮助的。

⊖ 为了与早期的C语言版本兼容，如果没有明确提供函数原型，编译器会假设它具有某些特性。在你自己的代码中，应该遵循本书的实例，为每个函数定义原型。

整个集合。如果该复合类型包含多层数据结构，那么可以使用更多的花括号来指定内部结构。

A.8 数据类型

C语言中的每一个值都与一个数据类型相关联，数据类型是用它的值域和一组操作来刻画的。这些数据类型分为两类：不再包含内部组成部分的原子类型以及包含其他类型值的复合类型。

A.8.1 原子类型

C语言中定义了下列原子类型：

- **整型** 基本的整数类型是int，它对应于一个有符号整数。表示整数的位数随机器的不同而不同，从而影响到int型的最大值。所有兼容ANSI的编译器必须允许int型至少有16位，也就是说它们的数值范围至少必须为－32 768至32 767。为了使编译器能够为一个整数分配更多或更少的空间，整型int可以用long和short加以修饰。此外，整型int、short和long可以单独被signed或unsighed修饰，表示是否允许负值。当使用修饰符时，可以省略类型名int。
- **浮点数** C语言定义了三种浮点类型：float、double和long double。这几种类型的精度和所需的存储空间取决于编译器和基本的硬件。对大部分应用来说，用double型就够了。
- **字符型** char型用来表示一个单独的字符。因为char型的值被定义为只需要一个字节的存储空间，因此字符型是表达数据值大小的基本单元。
- **枚举类型** 除了内置的原子类型外，C语言还允许你定义枚举类型，它们的值是有限个已命名的常量的集合。在本书中，用来引入一个新的枚举类型的标准句法是：

```
typedef enum {list of elements} type name;
```

659

接口genlib.h用枚举类型定义bool类型，定义方法如下：

```
typedef enum {FALSE, TRUE} bool;
```

每个枚举常量被赋予一个从0开始的整数。这样，在bool类型中，FALSE的内部值为0，TRUE的内部值为1。

A.8.2 数组

数组是一组存储在连续的存储单元中的有序的、同质的值。声明一个数组的标准句法是：

```
element type name[size];
```

在这个范例中，*element type*说明存储于数组的每个元素中的值的类型，*name*是数组变量的名字，*size*表示分配的元素个数。如果你不能确定要在数组中存放多少个元素，通常的方法是用一个大于你的应用所需的数作为*size*的值，然后只使用所分配空间的一部分。如果这样做的话，还需要声明一个整数来记录在某一时刻数组中有多少个有效元素。相对于分配长度，这个整数指出了数组的有效长度。

在一个数组中，每一个元素都用一个数字下标来标识。在C语言中，数组下标值总是从0开始。要从数组中选择一个特定元素，需要用一个选择表达式，如array[i]，其中数组名后跟一个括在方括号里的表达式，该表达式指出所要找的元素下标值。

A.8.3 指针

指针是一个数值的地址。要声明一个变量指向特定目标类型的值，C语言用下列声明形式：

```
目标类型 *名字;
```

A.9 ANSI库

本节描述了本书中用到的ANSI库函数，按接口来排列。

A.9.1 标准I/O库（stdio.h）

标准I/O库提供了一套操纵文件和执行输入/输出操作的函数，stdio.h接口通常被包含在每一个.C文件中。

• FILE *fopen(string filename, string mode);

该函数打开一个文件，并返回一个指向该文件的指针。文件的名字由字符串filename给出，字符串mode指定数据传输的类型，通常为下列各值之一：

- "r" 以读方式打开文件。文件必须已经存在。
- "w" 以写方式打开文件。如果文件存在，所有原来内容都被删除。
- "a" 以添加方式打开文件。如果文件存在，新输出的数据出现在文件末尾。

如果出现错误（如以读方式打开文件，但该文件不存在），则fopen返回NULL。

• void fclose(FILE *fp);

这个函数将fopen生成的文件指针作为参数，并关闭它。

• int getc(FILE *infile);
 int getchar(void);

getc函数读文件infile，并返回它的下一个字符。返回的值是一个整数（而非字符），因此结果类型可以包含一个特殊值EOF，用来表示文件结束。getchar函数功能相同，只是它总是从标准输入文件中读取它的输入。

• int ungetc(char ch, FILE *infile);

这个函数把字符ch推入infile指定的文件流，这样下一个从文件读取字符的函数就会把ch再次读出来。每次只有一个字符能被推回文件流。 662

• int putc(char ch, FILE *outfile);
 int putchar(char ch);

putc函数把一个字符ch写到outfile指定的输出文件中。putchar函数与之类似，只是它总是把字符写到标准输出文件。

• int fgets(char buffer[], int max, FILE *infile);

这个函数从输入文件中读入一行文本并把它们存入buffer中，buffer必须是一个由客户分配的字符数组。用这个函数最多可以读入max个字符，允许客户自己控制以防止缓冲区溢出。换行字符作为行结束标志也存储在缓冲区中。

• int fputs(string s, FILE *outfile);

这个函数把字符串s写入文件outfile中。

• void printf(string msg,. . .);
 void fprintf(FILE *outfile, string msg,. . .);
 void sprintf(char buffer[], string msg,. . .);

printf函数在标准输出设备上显示消息msg，该标准输出设备一般为终端屏幕，fprintf和sprintf函数类似于printf，只是它们输出到outfile或buffer。在这三种形式的任意一种中，msg中的字符被复制到输出流，如果在msg中出现一个百分号，它表示在该位置要用某个值替换。这个值为下一个未用的参数，这样，替代值在msg中出现的顺序将与在参数列表中出

此函数返回小于或等于x的最大整数的浮点表示。

• double ceil(double x);

此函数返回大于或等于x的最小整数的浮点型表示。

• double fmod(double x, double y);

此函数返回x/y的余数的浮点数。

• double sqrt(double x);

此函数返回x的平方根。

• double pow(double x, double y);

此函数返回x^y。

• double exp(double x);

此函数返回e^x。

• double log(double x);

此函数返回x的自然对数。

• double sin(double theta);

此函数返回角度为theta的正弦函数值，角度以弧度表示。

• double cos(double theta);

此函数返回角度为theta的余弦函数值，角度以弧度表示。

• double atan(double x);

此函数返回值x的反正切函数值，结果是$-\pi/2$到$+\pi/2$之间的角度，以弧度表示。

• int atan2(double y, double x);

此函数返回x轴与从原点到点（x, y）之间的直线所形成的角度。与其他三角函数一样，该角度以弧度表示。 665

A.9.4 字符类型库（ctype.h）

该接口输出一组极有效率的函数来分类不同的字符。[⊖]

```
• bool isalpha(char ch);
  bool isupper(char ch);
  bool islower(char ch);
  bool isdigit(char ch);
  bool isalnum(char ch);
  bool ispunct(char ch);
  bool isspace(char ch);
```

如果字符ch在指定的类别内，这些函数返回TRUE，如下所示：

```
isalpha  字母字符
isupper  大写字母字符
islower  小写字母字符
isdigit  十进制数字
isalnum  字母字符和数字的并集
```

⊖ ctype.h接口中的大部分函数定义为使用int类型，这样这些函数可以调用EOF常量。这里的原型使用char是为了增强可读性。

附录B 库源代码

此附录包含本书所用到的扩展库的接口和具体实现。这些库被设计成可移植的，可以在任何遵循ANSI标准的编译器上使用。

这些库的源代码以及本书中用到的所有程序实例的代码，可以登录www.hzbook.com来申请。

如果你有其他问题，还请写信到下列地址：

The Art & Science of C
Computer Science:Higher Ed Editorial
Addison-Wesley Publishing Company
One Jacob Way
Reading, MA 01867
U.S.A.

668
~
670

```
/*
 * File: genlib.h
 * --------------
 * This file contains several definitions that form the
 * core of a general-purpose ANSI C library developed by Eric
 * Roberts.  The goal of this library is to provide a basic
 * set of tools and conventions that increase the readability
 * of C programs, particularly as they are used in a teaching
 * environment.
 *
 * The basic definitions provided by genlib.h are:
 *
 *    1.  Declarations for several new "primitive" types
 *        (most importantly bool and string) that are
 *        used throughout the other libraries and
 *        applications as fundamental types.
 *
 *    2.  A new set of functions for memory allocation.
 *
 *    3.  A function for error handling.
 */

#ifndef _genlib_h
#define _genlib_h

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

/* Section 1 -- Define new "primitive" types */
```

图B-1 genlib.h

```
/* Section 2 -- Memory allocation */

/*
 * General notes:
 * --------------
 * These functions provide a common interface for memory
 * allocation.  All functions in the library that allocate
 * memory do so using GetBlock and FreeBlock.  Even though
 * the ANSI standard defines malloc and free for the same
 * purpose, using GetBlock and FreeBlock provides greater
 * compatibility with non-ANSI implementations, automatic
 * out-of-memory error detection, and the possibility of
 * substituting a garbage-collecting allocator.
 */

/*
 * Function: GetBlock
 * Usage: ptr = (type) GetBlock(nbytes);
 * -------------------------------------
 * GetBlock allocates a block of memory of the given size.  If
 * no memory is available, GetBlock generates an error.
 */

void *GetBlock(size_t nbytes);

/*
 * Function: FreeBlock
 * Usage: FreeBlock(ptr);
 * ----------------------
 * FreeBlock frees the memory associated with ptr, which must
 * have been allocated using GetBlock, New, or NewArray.
 */

void FreeBlock(void *ptr);

/*
 * Macro: New
 * Usage: p = New(pointer-type);
 * -----------------------------
 * The New pseudofunction allocates enough space to hold an
 * object of the type to which pointer-type points and returns
 * a pointer to the newly allocated pointer.  Note that
 * "New" is different from the "new" operator used in C++;
 * the former takes a pointer type and the latter takes the
 * target type.
 */

#define New(type) ((type) GetBlock(sizeof *((type) NULL)))

/*
 * Macro: NewArray
 * Usage: p = NewArray(n, element-type);
 * -------------------------------------
```

673

图 B-1 (续)

```
void *GetBlock(size_t nbytes)
{
    void *result;

    result = malloc(nbytes);
    if (result == NULL) Error("No memory available");
    return (result);
}

void FreeBlock(void *ptr)
{
    free(ptr);
}

/* Section 3 -- Basic error handling */

void Error(string msg, ...)
{
    va_list args;

    va_start(args, msg);
    fprintf(stderr, "Error: ");
    vfprintf(stderr, msg, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(ErrorExitStatus);
}
```

图 B-2 (续)

675

```
/*
 * File: simpio.h
 * --------------
 * This interface provides access to a simple package of
 * functions that simplify the reading of input data.
 */

#ifndef _simpio_h
#define _simpio_h

#include "genlib.h"

/*
 * Function: GetInteger
 * Usage: i = GetInteger();
 * ------------------------
 * GetInteger reads a line of text from standard input and scans
 * it as an integer.  The integer value is returned.  If an
 * integer cannot be scanned or if more characters follow the
 * number, the user is given a chance to retry.
 */

int GetInteger(void);

/*
 * Function: GetLong
```

图B-3 simpio.h

```
#include <stdio.h>
#include <string.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/*
 * Constants:
 * ----------
 * InitialBufferSize -- Initial buffer size for ReadLine
 */

#define InitialBufferSize 120

/* Exported entries */

/*
 * Functions: GetInteger, GetLong, GetReal
 * ---------------------------------------
 * These functions first read a line and then call sscanf to
 * translate the number.  Reading an entire line is essential to
 * good error recovery, because the characters after the point of
 * error would otherwise remain in the input buffer and confuse
 * subsequent input operations.  The sscanf line allows white space
 * before and after the number but no other extraneous characters.
 */

int GetInteger(void)
{
    string line;
    int value;
    char termch;

    while (TRUE) {
        line = GetLine();
        switch (sscanf(line, " %d %c", &value, &termch)) {
          case 1:
            FreeBlock(line);
            return (value);
          case 2:
            printf("Unexpected character: '%c'\n", termch);
            break;
          default:
            printf("Please enter an integer\n");
            break;
        }
        FreeBlock(line);
        printf("Retry: ");
    }
}

long GetLong(void)
{
    string line;
    long value;
    char termch;
```

677

678

图 B-4 (续)

```
 * ------------------
 * This function operates by reading characters from the file
 * into a dynamically allocated buffer.  If the buffer becomes
 * full before the end of the line is reached, a new buffer
 * twice the size of the previous one is allocated.
 */

string ReadLine(FILE *infile)
{
    string line, nline;
    int n, ch, size;

    n = 0;
    size = InitialBufferSize;
    line = GetBlock(size + 1);
    while ((ch = getc(infile)) != '\n' && ch != EOF) {
        if (n == size) {
            size *= 2;
            nline = (string) GetBlock(size + 1);
            strncpy(nline, line, n);
            FreeBlock(line);
            line = nline;
        }
        line[n++] = ch;
    }
    if (n == 0 && ch == EOF) {
        FreeBlock(line);
        return (NULL);
    }
    line[n] = '\0';
    nline = (string) GetBlock(n + 1);
    strcpy(nline, line);
    FreeBlock(line);
    return (nline);
}
```

图 B-4 (续)

680

```
/*
 * File: strlib.h
 * --------------
 * The strlib.h file defines the interface for a simple
 * string library.  In the context of this package, strings
 * are considered to be an abstract data type, which means
 * that the client relies only on the operations defined for
 * the type and not on the underlying representation.
 */

/*
 * Cautionary note:
 * ----------------
 * Although this interface provides an extremely convenient
 * abstraction for working with strings, it is not appropriate
 * for all applications.  In this interface, the functions that
 * return string values (such as Concat and SubString) do so
```

图B-5 strlib.h

```
 * 2. If p2 is greater than the index of the last string
 *    position, which is StringLength(s) - 1, then p2 is
 *    set equal to StringLength(s) - 1.
 * 3. If p2 < p1, SubString returns the empty string.
 */

string SubString(string s, int p1, int p2);

/*
 * Function: CharToString
 * Usage: s = CharToString(ch);
 * ----------------------------
 * This function takes a single character and returns a
 * one-character string consisting of that character.  The
 * CharToString function is useful, for example, if you
 * need to concatenate a string and a character.  Since
 * Concat requires two strings, you must first convert
 * the character into a string.
 */

string CharToString(char ch);

/*
 * Function: StringLength
 * Usage: len = StringLength(s);
 * -----------------------------
 * This function returns the length of s.
 */

int StringLength(string s);

/*
 * Function: CopyString
 * Usage: newstr = CopyString(s);
 * ------------------------------
 * CopyString copies the string s into dynamically allocated
 * storage and returns the new string.  This function is not
 * ordinarily required if this package is used on its own,
 * but is often necessary when you are working with more than
 * one string package.
 */

string CopyString(string s);

/* Section 2 -- String comparison functions */

/*
 * Function: StringEqual
 * Usage: if (StringEqual(s1, s2)) ...
 * -----------------------------------
 * This function returns TRUE if the strings s1 and s2 are
 * equal.  For the strings to be considered equal, every
 * character in one string must precisely match the
 * corresponding character in the other.  Uppercase and
 * lowercase characters are considered to be different.
 */
```

682

图 B-5 (续)

```
 * ---------------------------------
 * This function returns a new string with all
 * alphabetic characters converted to upper case.
 */

string ConvertToUpperCase(string s);

/* Section 5 -- Functions for converting numbers to strings */

/*
 * Function: IntegerToString
 * Usage: s = IntegerToString(n);
 * ------------------------------
 * This function converts an integer into the corresponding
 * string of digits.  For example, IntegerToString(123)
 * returns "123" as a string.
 */

string IntegerToString(int n);

/*
 * Function: StringToInteger
 * Usage: n = StringToInteger(s);
 * ------------------------------
 * This function converts a string of digits into an integer.
 * If the string is not a legal integer or contains extraneous
 * characters, StringToInteger signals an error condition.
 */

int StringToInteger(string s);

/*
 * Function: RealToString
 * Usage: s = RealToString(d);
 * ---------------------------
 * This function converts a floating-point number into the
 * corresponding string form.  For example, calling
 * RealToString(23.45) returns "23.45".  The conversion is
 * the same as that used for "%G" format in printf.
 */

string RealToString(double d);

/*
 * Function: StringToReal
 * Usage: d = StringToReal(s);
 * ---------------------------
 * This function converts a string representing a real number
 * into its corresponding value.  If the string is not a
 * legal floating-point number or if it contains extraneous
 * characters, StringToReal signals an error condition.
 */

double StringToReal(string s);

#endif
```

图 B-5 (续)

```
{
    int len;

    if (s == NULL) Error("NULL string passed to IthChar");
    len = strlen(s);
    if (i < 0 || i > len) {
        Error("Index outside of string range in IthChar");
    }
    return (s[i]);
}

string SubString(string s, int p1, int p2)
{
    int len;
    string result;

    if (s == NULL) Error("NULL string passed to SubString");
    len = strlen(s);
    if (p1 < 0) p1 = 0;
    if (p2 >= len) p2 = len - 1;
    len = p2 - p1 + 1;
    if (len < 0) len = 0;
    result = CreateString(len);
    strncpy(result, s + p1, len);
    result[len] = '\0';
    return (result);
}

string CharToString(char ch)
{
    string result;

    result = CreateString(1);
    result[0] = ch;
    result[1] = '\0';
    return (result);
}

int StringLength(string s)
{
    if (s == NULL) Error("NULL string passed to StringLength");
    return (strlen(s));
}

string CopyString(string s)
{
    string newstr;

    if (s == NULL) Error("NULL string passed to CopyString");
    newstr = CreateString(strlen(s));
    strcpy(newstr, s);
    return (newstr);
}

/* Section 2 -- String comparison functions */
```

687

图 B-6 (续)

```
    result[i] = '\0';
    return (result);
}

string ConvertToUpperCase(string s)
{
    string result;
    int i;

    if (s == NULL) {
        Error("NULL string passed to ConvertToUpperCase");
    }
    result = CreateString(strlen(s));
    for (i = 0; s[i] != '\0'; i++) result[i] = toupper(s[i]);
    result[i] = '\0';
    return (result);
}

/* Section 5 -- Functions for converting numbers to strings */

string IntegerToString(int n)
{
    char buffer[MaxDigits];

    sprintf(buffer, "%d", n);
    return (CopyString(buffer));
}

int StringToInteger(string s)
{
    int result;
    char dummy;

    if (s == NULL) {
        Error("NULL string passed to StringToInteger");
    }
    if (sscanf(s, " %d %c", &result, &dummy) != 1) {
        Error("StringToInteger called on illegal number %s", s);
    }
    return (result);
}

string RealToString(double d)
{
    char buffer[MaxDigits];

    sprintf(buffer, "%G", d);
    return (CopyString(buffer));
}

double StringToReal(string s)
{
    double result;
    char dummy;

    if (s == NULL) Error("NULL string passed to StringToReal");
```

689

690

图 B-6 (续)

```
/*
 * Function: RandomReal
 * Usage: d = RandomReal(low, high);
 * ---------------------------------
 * This function returns a random real number in the half-open
 * interval [low .. high), meaning that the result is always
 * greater than or equal to low but strictly less than high.
 */

double RandomReal(double low, double high);

/*
 * Function: RandomChance
 * Usage: if (RandomChance(p)) . . .
 * ---------------------------------
 * The RandomChance function returns TRUE with the probability
 * indicated by p, which should be a floating-point number between
 * 0 (meaning never) and 1 (meaning always). For example, calling
 * RandomChance(.30) returns TRUE 30 percent of the time.
 */

bool RandomChance(double p);

#endif
```

692

图 B-7 (续)

```
/*
 * File: random.c
 * --------------
 * Implements the random.h interface.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "genlib.h"
#include "random.h"

/*
 * Function: Randomize
 * -------------------
 * This function operates by setting the random number
 * seed to the current time. The srand function is
 * provided by the <stdlib.h> library and requires an
 * integer argument. The time function is provided
 * by <time.h>.
 */

void Randomize(void)
{
    srand((int) time(NULL));
}
```

693

图B-8 random.c

索　引

索引中的页码为英文原书页码，与书中边栏页码一致。

! operator (!运算符), 107
#define, 89, 649
#include, 23, 649
% (in printf) (% (在printf中)), 81
% remainder operator (%余数运算符), 45
%% specification (%%说明), 82
%d format (%d 格式), 82
%e format (%e 格式), 82
%f format (%f 格式), 82
%g format (%g 格式), 82
%s format (%s 格式), 82
& operator (&运算符), 458
&& operator (&&运算符), 107
* operator (*运算符), 43, 458
= operator (=运算符), 62
+ operator (+运算符), 43
++ operator (++运算符), 63, 473
+= operator (+=运算符), 62
- operator (-运算符), 43
-- operator (--运算符), 63, 473
-= operator (-=运算符), 62
-> operator (一运算符), 570
. operator (.运算符), 561
/ operator (/运算符), 43
/= operator (/=运算符), 62
; (statement terminator) (; (语句结束符)), 104
?: operator (?: 运算符), 115～117
|| operator (||运算符), 107
\0, 参见null character
\n, 参见newline

A

abacus (算盘), 2
abs, 145, 664
absolute coordinates (绝对坐标), 226
abstract data type (ADT) (抽象数据类型 (ADT)), 319, 615
abstraction (抽象), 224
abstraction boundary (抽象边界), 262
acronym (取首字母的缩写词), 326
Ada, 3
add10.c, 70
add2.c, 26
add2f.c, 35
addlist.c, 72, 124
AddPoint, 563
address (地址), 383
AdjustPen, 239
ADT, 参见abstract data type
adventure games (冒险游戏), 589
Aesop, 261
Alfonso the Wise, 425
algorithm (算法), 9, 185～213, 445, 628
algorithmic design (算法设计), 10
algorithmic efficiency (算法效率), 189, 628
alignment (对齐), 84
allocated size (分配长度), 391
allocation (分配), 385, 478～482
anagram (由颠倒字母顺序而构成的字), 611
analysis of algorithms (算法分析), 445, 628
Analytical Engine (分析机), 2
ANSI C (ANSI C), 15
ANSI string library (ANSI字符串库), 502
answerT, 579
argument (参数), 25, 139
arithmetic expression (算术表达式), 43
arithmetic overflow (算术溢出), 272
array (数组), 375～414
array element (数组元素), 377
array selection (数组选择), 378
array size (数组长度), 377
ASCII, 309
assignment statement (赋值语句), 28, 41
associativity (结合性), 103
atan, 665
atan2, 665
Atanasoff, John, 3
AtEndOfLine, 361
atomic data type (原子数据类型), 106, 659
automatic allocation (自动分配), 478
automatic type conversion (自动类型转换), 48
ave2f.c, 47

cursor（光标），8
cycle（周期），67，120
cyclic cipher（循环加密），336

D

dangling-else problem（悬空else问题），113
data（数据），25
data structure（数据结构），376
data type（数据类型），33
data-driven（数据驱动），588
database（数据库），570
De Morgan's law（德·摩根定律），109
debugging（调试），13，76～80，190～192，270～272
declaring a variable（声明一个变量），27，40
decomposition（分解），165～176
decrement operator（自减运算符），63，473
default clause（default子句），118
default value（缺省值），359
Dequeue，622
dequeue operation（出队操作），616
dereferencing a pointer（间接引用一个指针），458
desk-checking（桌面检查），189
dicetest.c，279
Dickens，Charles，558
dict.h，598
Difference Engine（差分机），2
digitsum.c，122
Dijkstra Edsger，450
directionT，306
disk（磁盘），8，524
divide-and-conquer（分而治之），631
do statement（do语句），655
dot operator（点运算符），561
double，34，212
DrawArc，233
DrawBox，237
drawbox.c，238
DrawCenteredCircle，237
DrawHouse，243
DrawLine，231
DrawLineTo，239
DrawPolarLine，643
dynamic allocation（动态分配），478～482
dynamic array（动态数组），480
dynamic initialization（动态初始化），357

E

Eckert，J. Presper，3
effective size（有效长度），391
efficiency（效率），189，628～637
Einstein，Albert（阿尔伯特·爱因斯坦），263
element type（元素类型），377
elements.c，545
else clause（else子句），113
embedded assignment（赋值嵌套），102
employeeDB，570
empty string（空字符串），322
encapsulation（封装），576
encoding（编码），304
end-of-file marker（文件结束标志），525
ENIAC，3
Enqueue，620
enqueue operation（入队操作），616
Enter key（Enter键），9
enum，305
enumeration type（枚举类型），303
EOF，529
Error，200
error handling（错误处理），201
escape sequence（转义序列），311，650
Euclid，195
Euclid's algorithm（Euclid算法），195
executable file（可执行文件），11
exp，665
exporting a function（导出一个函数），223
expression（表达式），37～51，652～653
extending an interface（扩展一个接口），264
extension（扩展），11

F

fabs，182，665
fact.c，152
Factorial，147，604
FALSE，105
fclose，528，662
feature enhancement（功能增强），14
fgets，536，663
field（字段），559
field width（字段宽度），84
file（文件），11，523～548
FILE *，526
file transfer protocol（FTP）（文件传输协议（FTP）），670
fill.h，372
filling（填充），371
FindChar，325
findcoin.c，427
FindFirstVowel，352，495～496
FindIntegerInArray，427

IntegerToString, 329
integrated circuit (集成电路), 4
interface (接口), 221～251, 259～290
interface entry (接口项), 275
internal state (内部状态), 349
Internet worm (因特网蠕虫), 507
invert.c, 512
isalnum, 314, 666
isalpha, 314, 666
isdigit, 314, 666
IsEven, 148
IsLegalWord, 350
islower, 314, 666
IsPrime, 187, 192
ispunct, 314, 666
isspace, 314, 666
isupper, 314, 666
iteration (迭代), 67, 105
IthChar, 321

J

Jacquard. Joseph Marie, 3
Juster, Norton, 301

K

Kernighan, Brian, 22
keyword (关键词), 39
Koch snowflake (Koch雪花), 642
Koch, Helge von, 642

L

Lao-tzu (老子), 261
layered abstraction (分层抽象), 317～319, 502
leapyear.c, 112
left alignment (左对齐), 84
left-associative (左结合), 103
Leibniz, Gottfried, 2
letter-substitution cipher (字母替代加密), 336
lexical analysis (词法分析), 348
lexicographic order (词典顺序), 324
library (库), 11, 23, 138～140, 221～251, 274～283, 662～667
liftoff.c, 127
line drawing (画线), 231
line graph (折线图), 420
linear search (线性搜索), 431～432
linear time (线性时间), 630
linked list (链接表), 628
linking (链接), 12
ListEmployees, 565, 574
ListPermutations, 613
local variable (局部变量), 154, 478
log, 665
logarithm (对数), 435, 635, 665
logical and (逻辑与), 107
logical not (逻辑非), 107
logical operator (逻辑运算符), 107
logical or (逻辑或), 107
long, 211
long double, 212
loop (循环), 67, 119～130
loop invariant (循环不变式), 188
loop-and-a-half problem (循环半途退出问题), 123
Lord Acton, 15
Lord Byron, 3
lvalue (左值), 455

M

main program (主程序), 24
main module (主模块), 341
malloc, 478, 664
Mandelbrot, Benoit, 641
math library (数学库), 139
math.h, 139, 664
matrix (矩阵), 410
Mauchly, John, 3
mean (均值), 417
median (中值), 448
member (成员), 559
memory (内存), 7, 382～388, 478～482
memory allocation (内存分配), 478～482
Merge, 634
merge sort (合并排序), 633～637
merging (合并), 632
microprocessor (微处理器), 4
MinF, 145
mnemonic (助记词), 644
mode (模式), 449
modular arithmetic (取模运算), 169
modular development (模块化开发), 339～366
module (模块), 340
Monte Carlo integration (蒙特卡洛积分), 295
MonthName, 147
Morse code (Morse码), 419
MovePen, 231
multidimensional array (多维数组), 410～413
multiple assignment (多重赋值), 102

RandomReal, 284
randtest.c, 267
RAND_MAX, 266
rational number (有理数), 594
read-an-integer idiom (读入一个整型数习语), 61
read-until-sentinel idiom (读入一直到一标志习语), 71
ReadEmployeeDatabase, 573
ReadLine, 538
ReadOneEmployee, 573
RealToString, 329
record (记录), 557～589
record selection (记录选择), 561
recursion (递归), 602～614
recursive leap of faith (递归信任), 610
reductionism (归约论), 32
relational operator (关系运算符), 75, 105
relative coordinates (相对坐标), 226
remainder operator (%) (取余运算符 (%)), 45
remove, 664
rename, 664
repeat-N-times idiom (重复N次习语), 65
repfirst.c, 328
return address (返回地址), 160
Return key (Return键), 9, 28
return statement (return语句), 142, 655
returning a value (返回一个值), 28, 139
reverse.c, 399
Reynolds, Malvina, 375
right alignment (右对齐), 84
right-associative (右结合), 103
right-hand rule (右手规则), 214
ring buffer (环缓冲区), 627
Ritchie, Dennis, 15, 22
root name (根名), 11

S

sample run (运行示例), 9
scalar type (标量类型), 307
scanf, 539－547, 663
scanner.c, 361
scanner.h, 353
scanning (扫描), 348
Schickard, Wilhelm, 2
scientific notation (科学记数法), 38
scope (范围), 355
searching (查找), 426～437
secondary storage (辅助存储器), 8
seed (种子), 282
selection (选择), 378, 561
selection expression (选择表达式), 378
selection sort (选择排序), 438～446
semantics (语义), 188
semicolon (分号), 104
sentinel (标志), 70
series expansion (级数展开), 201～209
Shakespeare, William (威廉·莎士比亚), 259, 491
short, 211
short-circuit evaluation (简化求值), 109
shorthand assignment (复合赋值运算符), 62
signtest.c, 115
simpio.c, 677
simpio.h, 26, 676
simple statement (简单语句), 100, 653
sin, 139, 665
sizeof, 385
Smith, Al, 557
Software (软件), 6
software engineering (软件工程), 14
software maintenance (软件维护), 13
sort.c, 443
sort.h, 439
sorting (排序), 437－446
SortIntegerArray, 441, 634
source file (源文件), 11
special character (特殊字符), 311
sprintf, 539, 663
sqrt, 139, 665
srand, 282, 664
sscanf, 540, 663
stack frame (栈帧), 154
standard deviation (标准差), 417
standard files (标准文件), 528
statement (语句), 25
static, 356
static allocation (静态分配), 478
static initialization (静态初始化), 357, 404～410, 561～562
stderr, 528
stdin, 528
stdio.h, 24, 525, 662
stdlib.h, 145, 266, 478, 664
stdout, 528
Stein, Gertrude, 523
stepwise refinement (逐步精化), 165～176
Stoppard, Tom, 293
strcat, 508, 667
strchr, 510, 667
strcmp, 509, 666

本书是一本计算机科学的经典教材，强调软件工程和优秀的程序设计风格。此外，学生还可以从书中学习到 ANSI C 的基础知识，这些内容已经成为计算机行业的标准。作者的写作风格使得书中深奥的概念变得易于理解和引人入胜。

本书集中讨论库和抽象的用法，这是当代程序设计技术中最基本的知识。作者使用库来隐藏C语言的复杂性，更加突出主题，使学生可以较好地掌握每一个主题的精髓。然后，进一步给出每个库的底层实现，较好地展示了库自身的抽象威力。

本书从基础开始讲起，是C语言的入门教材。本书已经被美国斯坦福大学、哥伦比亚大学等多所大学和学院成功采用，在国内已被上海交通大学采用，是一本适合高等院校计算机及相关专业使用的优秀教材。

读者可登录http://www.hzbook.com下载本书的源代码和其他资源。

作者简介

Eric S. Roberts 美国斯坦福大学计算机科学系教授，也是该系主管教学事务的副主任，同时他还是工学院的Charles Simonyi讲席教授。他于1980年在哈佛大学应用数学系获得博士学位，并曾在DEC公司位于加州Palo Alto的系统研究中心做过5年的研究工作。作为一位获得高度评价的教育工作者，Roberts还因其在本科生教学中的杰出贡献获得了1993年的Bing Award奖。他的另一本书《C程序设计的抽象思维》(Programming Abstractions in C: A Second Course in Computer Science) 的英文影印版已由机械工业出版社引进出版。





《C语言的科学和艺术》(英文版)
ISBN 7-111-13991-7 定价：60.00 元

www.PearsonEd.com

ISBN 7-111-15971-3

封面设计 陈子平

HZ BOOKS 华章图书

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

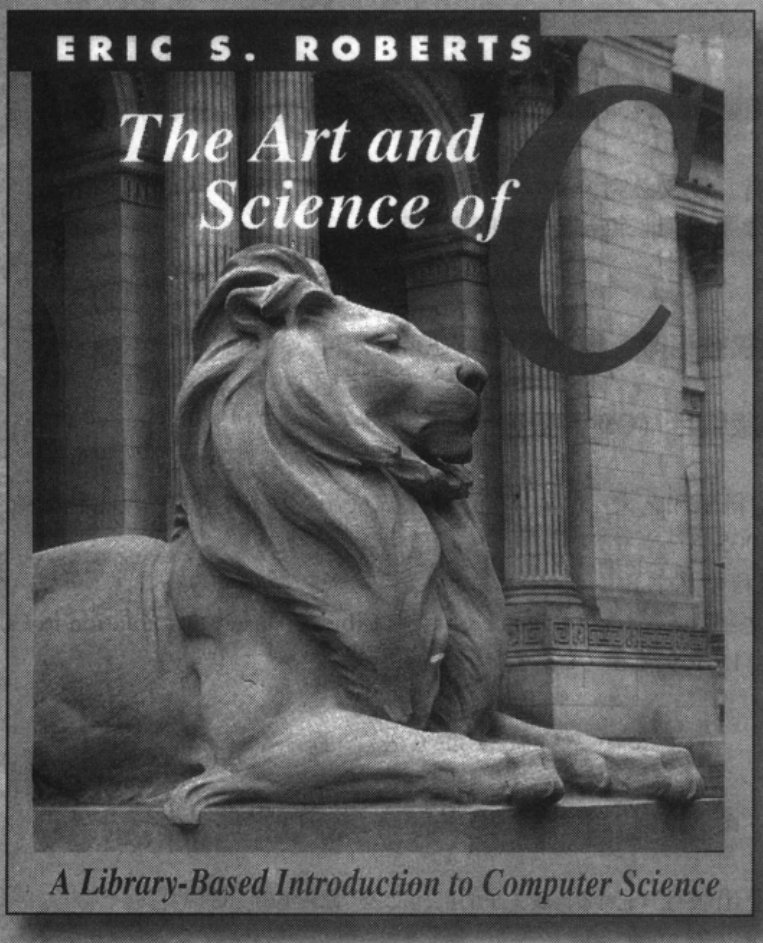
北京市西城区百万庄南街1号 100037
读者服务热线：(010)68995259, 68995264
读者服务信箱：hzedu@hzbook.com

ISBN 7-111-15971-3/TP · 4160
定价：55.00 元

计 算 机 科 学 丛 书

C语言的科学和艺术

(美) Eric S. Roberts 斯坦福大学 著　翁惠玉 张冬茉 杨鑫 蒋文新 译



The Art and Science of C
A Library-Based Introduction to Computer Science

机械工业出版社
China Machine Press

本书是计算机科学的经典教材，介绍了计算机科学的基础知识和程序设计的专门知识。本书以介绍ANSI C为主线，不仅涵盖C语言的基本知识，而且介绍了软件工程技术以及如何应用良好的程序设计风格进行开发等内容。本书采用了库函数的方法，强调抽象的原则，详细阐述了库和模块化开发。此外，本书还利用大量实例讲述解决问题的全过程，对开发过程中常见的错误也给出了解决和避免的方法。本书既可作为高等院校计算机科学入门课程及C语言入门课程的教材，也是C语言开发人员的极佳参考书。

Simplified Chinese edition copyright © 2005 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *The Art and Science of C: A Library-Based Introduction to Computer Science* (ISBN: 0-201-54322-2) by Eric S. Roberts, copyright © 1995.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

版权所有，侵权必究。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2004-1204

图书在版编目（CIP）数据

C语言的科学和艺术/（美）罗伯茨（Roberts, E. S.）著；翁惠玉等译. —北京：机械工业出版社，2005.3

（计算机科学丛书）

书名原文：The Art and Science of C: A Library-Based Introduction to Computer Science

ISBN 7-111-15971-3

Ⅰ.C… Ⅱ.①罗… ②翁… Ⅲ.C语言—程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2005）第000915号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：朱 劼

北京昌平奔腾印刷厂印刷·新华书店北京发行所发行

2005年3月第1版第1次印刷

787mm×1092mm 1/16·32.5印张

印数：0 001-4 000册

定价：55.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

专家指导委员会

（按姓氏笔画顺序）

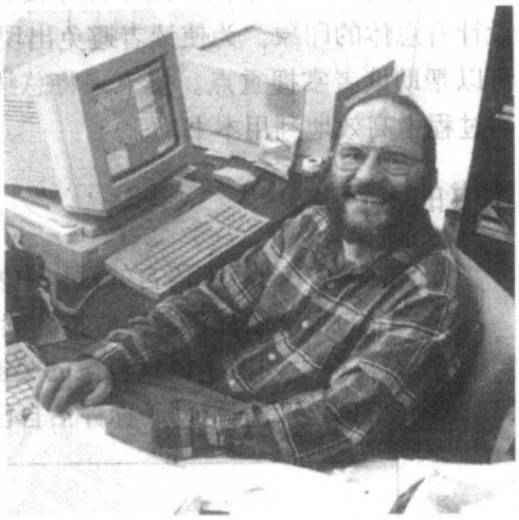
尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

秘　书　组

武卫东	温莉芳	刘　江	杨海玲

前　言

作者简介



Eric S. Roberts第一次教授计算机科学入门课程是在20多年前，当时他还是一名哈佛学子。1980年获得博士学位以来，Roberts先后在哈佛、韦尔斯利、斯坦福大学教授计算机科学课程。在斯坦福大学，Roberts担任计算机科学系副主任，并负责本科生的计算机科学课程。虽然Roberts也教授计算机科学的高级课程并作一些研究工作，但他最大的乐趣还是向初学者展示计算机的强大功能。

在斯坦福大学任教的同时，Roberts自1990年起还担任计算机专业协会主席。这是一个由计算机专业人士组成的公益团体，在全美有22个分会，2000名会员。计算机在多方面影响着我们的社会。学习有关计算机的技术固然重要，但确保计算机服务于社会也很重要。

若读者发现本书中有表述不清楚的地方或错误之处，恳请不吝赐教。请通过Roberts的电子邮箱ericr@aw.com与他联系。

致学生

欢迎你！拿起这本书，你就迈进了计算机科学的世界——这门学科出现在半世纪以前，现在却成为这个时代最具生机和活力的学科之一。

在几十年的发展过程中，计算机几乎使所有领域中看似不可能的事情成为可能。由于计算机可在瞬间将信息传递到任何地方，所以今天的企业家能以空前的规模经营跨国公司。由于计算机可进行必要的、但人工很难完成的计算，科学家才能解决许多问题。电影人利用计算机制作出更具感染力的视觉效果。由于计算机能处理医学中大量的信息处理，因此医生能对患者的病情做出更精确的诊断。

计算机技术正在飞速发展。目前我们已经看到的优势与新的世纪将要经历的发展相比肯定将相形见绌。最近50年，计算机已经对世界产生了深远影响，在新的世纪亦将如此。今日的学生将会是执行这项伟大的工程的中流砥柱。要做到这一点，就必须懂得如何使用计算机。

和其他值得掌握的技能一样，理解计算机的工作原理以及学会怎样控制它们是需要花费时间的。这一切不可能一蹴而就，必须从某个起点开始循序渐进。2500年前，中国的哲学家老子曾说过："千里之行，始于足下"。本书就是一个很好的起点。

然而对很多人来说，万事开头难。许多学生在计算机面前束手无策，认为计算机科学超出了他们的理解范围。可是基本的程序设计并不需要具备高等数学和电子学的知识。在程序

• 小结部分详细地描述了你应该学到的与学习目标部分相关的知识。

小结

第3章从宏观角度展示了程序设计的过程，强调了问题的求解。在此过程中，以非正式的方法介绍了一些控制语句。本章详细地研究了这些控制语句的工作细节。此外，还介绍了一个新的数据类型——布尔型数据。尽管这种数据类型只有TRUE和FALSE两个值，但是能高效地使用它对成功的程序设计十分重要，因此应该多加练习。

本章中还介绍了几个新的运算符，在这里回顾一下已学过的运算符间的优先级关系是有好处的。表4-1总结了这些信息，所有的运算符是按优先级递减的顺序列出的。

表4-1 本章中出现过的运算符的优先级表

运 算 符	结 合 性
一元 ++ -- ! (类型转换)	从右到左
* / %	从左到右
+ -	从左到右
< <= > >=	从左到右
== !=	从左到右
&&	从左到右
\|\|	从左到右
?:	从右到左
op=	从右到左

本章的重要知识点包括：

- 简单语句由表达式跟上分号构成。
- “=”是C语言中用来赋值的运算符。因此，赋值是一个合法的的表达式。它可以以嵌套赋值和多重赋值的形式出现。
- 多个独立的语句可以组合为复合语句，通常称为程序块。
- 控制语句分成两类：条件和迭代。
- genlib库定义了一种bool数据类型，用它来表示布尔型数据。bool型数据只有TRUE和FALSE两个值。
- 使用关系运算符（<、<=、>、>=、==和!=）能够产生布尔值，然后可以使用逻辑运算符（&&、||和!）连接关系表达式。
- 逻辑运算符&&和||是从左到右计算的，一旦能确定整个表达式的值就停止计算。这种行为叫做简化求值。
- 当有些代码仅在特定情况下才执行或程序需要在两条执行路径之间选择其一时可以使用if语句。
- 有些问题有这样的结构：在情况1下完成操作1，情况2下完成操作2，依此类推。此时就可以使用switch语句。
- while语句用来指定在一定条件满足的情况下重复执行某些操作。
- for语句指示一定次数的重复操作，它在每个周期中要更新下标变量的值。

2) 程序设计既是一门科学，也是一门艺术。形成良好的程序设计风格需要掌握很多知识，而不只是记住一组规则。你必须通过实践并阅读其他程序来不断学习。书中在介绍程序设计时的一些特色可以为你提供帮助。

- 本书包括大量的程序实例，这些实例说明了如何用C语句建立一个完整的程序。这些实例也可作为你自己设计程序时的模型。在许多情况下，你可以对书中的程序作简单的修改来解决一个新的程序设计问题。

- 为了帮助你识别和纠正逻辑错误，本书包括了一些说明某些典型错误的程序。为了确保你不会把这些程序作为模型，本书在这些不正确的程序上面叠印了一个苍蝇的图像。



```
/*
 * File: balance2.c
 * ----------------
 * This file contains a buggy second attempt at a program to
 * balance a checkbook.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double entry, balance;

    printf("This program helps you balance your checkbook.\n");
    printf("Enter each check and deposit during the month.\n");
    printf("To indicate a check, use a minus sign.\n");
    printf("Signal the end of the month with a 0 value.\n");
    printf("Enter the initial balance: ");
    balance = GetReal();
    while (TRUE) {
        printf("Enter check (-) or deposit: ");
        entry = GetReal();
        if (entry == 0) break;
        balance += entry;
        if (balance < 0) {
            printf("This check bounces.  $10 fee deducted.\n");
            balance -= 10;
        }
        printf("Current balance = %g\n", balance);
    }
    printf("Final balance = %g\n", balance);
}
```

- 常见错误对所有初级程序员常犯的错误给出了提示，并说明如何避免这些错误。代码中的错误行用苍蝇图案加以突出，旁边给出相应的说明。

在大多数情况下，逻辑表达式不至于复杂到必须用真值表来计算的程度。唯一容易引起混淆的是!或!=和&&或||一起出现的情况。当说起某某情况不为真时（即需要用到!或!=的情况），日常用语和数学逻辑表达有时是相悖的，因此需要格外小心避免犯错。比如，在程序中要表达这样一个意思：*x*不等于2或3。通过这个条件测试语言表述，新程序员很可能会写出以下语句：



```
if (x != 2 || x != 3) . . .          (这是错误的)
```

如果用数学观点来观察这个条件测试，会发现只要x不等于2或只要x不等于3，if测试中的表达式均为TRUE。无论x取什么值，其中一个表达式必定为真，因为如果x为2，则它不可能同时为3，反之亦然。因此上面写出的if测试将永远成功。

常见错误 当&&和||运算符与涉及!和!=运算符的关系测试一起使用时，必须非常小心。自然语言在逻辑上有一些模糊，而程序设计要求非常精确。

- 早日学习C语言为学习C++和面向对象的语言铺平了道路。由于学生在第一年的学习后，就有了较丰富的使用C语言的经验，学校就可以更早开设教学计划中的面向对象的系统设计这门课程。
- C语言中覆盖了一些重要的主题，如模块化开发和抽象数据类型等。这在Pascal语言环境里是很难讲授的。
- 在最近五年中，在工程科学领域的编程中，C语言正在逐步取代Fortran的重要地位。

考虑到近三年的经历，作者确信选用C语言作为计算机科学的入门课程是正确的选择，这种改变将使将来的程序更为强壮。

与此同时，也要意识到在程序设计的早期课程中教授C语言并不是那么容易。C语言是为程序员设计的，并非为作为初学者的学生设计的。其中很多的特性只有在理解了大的概念框架后才有意义，而初学者往往意识不到这一点。从许多方面讲，C语言都是一门复杂的语言，向初学者讲授这门语言时必须很好地控制讲授的进度。

基于库函数的方法

让教师能够控制C语言所固有的复杂性是本教材的中心目标之一。而控制复杂性又是程序员的重要职责。当面对一个很复杂的、不能立即解决的问题时，可将它分解成多个部分，然后单独思考每个部分的解决方案。此外，当某个部分达到了一定的复杂度时，可以将它抽象出来，定义为一个简单的接口。这样，用户就不必仔细理解这个抽象的细节，从而简化了程序的概念结构。

这种方法也适用于程序设计的教学。为使知识更浅显易懂，本书采用了基于库函数的方法，该方法强调抽象的原则。这种方法的特征反映在以下两方面，这也是本书区别于其他入门教材之处：

1) 在开始部分详细地介绍库和模块化开发，这是现代程序设计的基本概念。第二部分集中介绍库、接口、抽象和模块化开发的知识。这样，学生在学习数组之前就可以先学会高效使用这些技术。

2) 本书通过库函数的使用来展示它们的功能。一方面告诉学生库函数可以隐藏复杂性，另一方面向他们说明了这个概念。本书介绍了一些新的库函数以帮助学生写出一些有用的程序，但在他们能消化吸收之前并不讲述库函数所隐藏的细节内容。它们的具体实现放在后面的章节讲述，以使学生能够更深刻地意识到抽象的重要作用。

在1992年，作者曾试图只用ANSI C的库函数来讲授入门课程，但结果并不理想。每个新的主题都要求学生对C语言的其余部分很熟悉，这样便无法开展教学。例如，字符串操作尽管在概念上很简单，但在学生能灵活使用字符串之前，他们必须理解数组、指针和内存分配等机制。最优秀的学生也是到了学期末才理解，而大部分学生则根本没有明白这些概念。自从1993年初引进基于库函数的方法以来，学生理解起来更容易了，对计算机这个学科的了解也更多了。

本书使用的库函数接口和相关实现见附录B。附录B还给出了获取源代码的方法。

讲述顺序

本书的章节顺序与斯坦福大学的入门课程的授课顺序大致相同，只是将第17章的内容放

驱动的教学程序，它综合运用了这些基本的数据结构，是本书中最复杂的一个程序设计示例。

第17章包含了在第一门程序设计课程中经常出现的三个重要主题：递归、抽象数据类型和算法分析。在斯坦福，这些主题在第二门课程中用1/4学期的时间讲述。若决定在第一门程序设计课程中讲述递归，强烈建议早日开始介绍，以便学生有时间去消化吸收。教师可以在第5章后直接讲述递归函数，在第6章后讲述递归算法，也可以在第12章后将递归和算法分析放在一起讲解。

教师资源

教师要获取本书的相关教学资源，请到华章网站www.hzbook.com上申请。

致谢

本书的前身是课程讲义，在此基础上进行了大量改进而形成本书，这很大程度上要归功于使用本书的读者的建议。在这里我要衷心感谢斯坦福大学的讲师们：Jon Becker、Katie Capps、Stephen Clausing、Todd Feldman、Allison Hansen、Margaret Johnson、Jim Kent、Andrew Kosoresow、Mehran Sahami和Julie Zelenski，他们在过去三年的14个班讲授了本书。此外，我还要感谢所有的班导师、助教和教务人员：Felix Baker、John Lilly、Sandy Nguyen、Bryan Rollins以及Scott Wiltamuth，他们提供了必要的后勤支持。

本书中很多好的想法来自于一个计算机入门教学的社团，这是我于1992～1993年倡导成立的。它为解决一些困难问题和对教材进行重要改进提供了一个讨论的平台。在此我想向所有参与其中的人表示感谢：Perry Arnold、Jon Becker、Tom Bogart、Karl Brown、Bryan Busse、Katie Capps、Peter Chang、Scott Cohen、Stacey Doerr、Jeff Forbes、Stephen Freund、Jon Goldberg、Tague Griffith、Matt Harad、Lindsay Lack、Christine Lee、Tricia Lee、John Lilly、Albert Lin、Mara Mather、Hugh McGuire、Jeffrey Oldham、David O'Keefe、Bryan Rollins、Samir Saxena、Nikhyl Singhal、Eric Tucker、Garner Weng、Howard Wong-Toi和John Yong。

同时，我还要感谢所有使用本书并为本书的初稿提出意见的学生们：Adjo Amekudzi、Eric Anderson、Andrew Arnold、Kevin Berk、Kevin Carbajal、Ajit Chaudhari、Alida Cheung、Hye-won Choi、Elisabeth、Christensen、Ralph Davis、Joel Firehammer、Peter Frelinghuysen、Trevor Gattis、Teddy Harris、Heather Heal、Enoch Huang、Ann Lee、Ted Lee、Daphne Lichtensztajn、Lisa Maddock、Allan Marcus、Brad McGoran、Forrest Melton、Adrienne Osborn、Masatoshi Saito、Anne Stern、Ricardo Urena和Nichole Wilson。

在1993～1994年，几位来自其他大学的教师试用了本书的初稿并提出了一些很有价值的建议。我要特别感谢哈佛大学的Margo Seltzer，内华达大学里诺分校的Rob Langsner，马里兰大学（巴尔的摩州）的Richard Chang，拉萨尔大学的Jane Turk和维斯特蒙德学院的Kim Kihlstrom。他们为本书早期阶段的修改提供了大力帮助。

我还要感谢对本书进行审校的人员：

Stephen Allan	犹他州立大学
James Schmolze	塔夫茨大学
Don Goelman	Villanova大学
Michael Skolnick	Rensselaer Polytechnic
Stan Kolasa	Rutgers大学

目　录

出版者的话
专家指导委员会
译者序
前言

第1章　概述 ……1
1.1　计算简史 ……1
1.2　什么是计算机科学 ……4
1.3　计算机硬件简介 ……4
1.3.1　CPU ……4
1.3.2　内存 ……5
1.3.3　辅助存储器 ……5
1.3.4　I/O设备 ……5
1.4　算法 ……6
1.5　程序设计语言和编译 ……7
1.6　编程错误和调试 ……8
1.7　软件维护 ……9
1.8　软件工程的重要性 ……10
1.9　关于C程序设计语言的一些思考 ……10
小结 ……10
复习题 ……11

第一部分　C语言程序设计基础

第2章　通过例子学习 ……14
2.1　“Hello world”程序 ……14
2.1.1　注释 ……15
2.1.2　库包含 ……15
2.1.3　主程序 ……16
2.2　两个数的加法程序 ……17
2.2.1　输入阶段 ……19
2.2.2　计算阶段 ……20
2.2.3　输出阶段 ……20
2.3　有关程序设计过程的观点 ……21
2.4　数据类型 ……22
2.4.1　浮点型数据 ……22
2.4.2　字符串类型的数据 ……23
2.5　表达式 ……25
2.5.1　常量 ……25
2.5.2　变量 ……26
2.5.3　赋值语句 ……27
2.5.4　运算符和操作数 ……29
2.5.5　整型数和浮点型数的结合 ……30
2.5.6　整数除法和求余运算符 ……30
2.5.7　优先级 ……31
2.5.8　优先级法则的应用 ……33
2.5.9　类型转换 ……33
小结 ……35
复习题 ……36
程序设计练习 ……37
第3章　问题求解 ……40
3.1　程序设计习语和范例 ……40
3.1.1　复合赋值习语 ……41
3.1.2　自增和自减运算符 ……42
3.2　解决规模稍大的问题 ……43
3.3　控制语句 ……44
3.3.1　重复N次习语 ……44
3.3.2　迭代和循环 ……45
3.3.3　下标变量 ……45
3.3.4　初始化的重要性 ……47
3.3.5　读入一直到一标志习语 ……47
3.3.6　创造一个更实用的应用程序 ……49
3.3.7　条件执行和if语句 ……51
3.4　一个调试练习 ……53
3.5　格式化输出 ……56
3.5.1　printf的格式码 ……57
3.5.2　控制空格、对齐方式和精度 ……57
3.6　构思一个程序 ……60
3.6.1　程序设计风格 ……60
3.6.2　设计时考虑将来的修改 ……61
3.6.3　#define机制 ……61

6.4.3 估计平方根的泰勒级数展开（可选）……143
6.4.4 泰勒级数近似的实现 ……144
6.4.5 停留在收敛半径之内 ……146
6.5 指定数值类型的大小……147
6.5.1 整数类型 ……148
6.5.2 无符号类型 ……149
6.5.3 浮点类型 ……149
小结 ……149
复习题 ……150
程序设计练习 ……150

第二部分 库和模块化开发

第7章 库和接口：一个简单的图形库 …156
7.1 接口的概念 ……156
7.2 图形库介绍 ……158
7.2.1 graphics.h的基本模型 ……158
7.2.2 graphics.h接口的函数 ……159
7.2.3 软件包初始化 ……163
7.2.4 画直线……163
7.2.5 画圆和弧 ……164
7.2.6 获取有关图形窗口的信息 ……166
7.3 建立自己的工具 ……166
7.3.1 定义DrawBox ……167
7.3.2 定义DrawCenteredCircle ……169
7.3.3 绝对坐标和相对坐标间的切换……169
7.3.4 定义过程的好处 ……170
7.4 解决一个较大的问题……170
7.4.1 使用逐步精化 ……171
7.4.2 实现DrawHouse过程 ……172
7.4.3 寻找共同的模式 ……172
7.4.4 结束分解……173
小结 ……177
复习题 ……178
程序设计练习 ……179
第8章 设计接口：一个随机数库 ……183
8.1 接口设计 ……183
8.1.1 同一主题的重要性 ……184
8.1.2 简单性和信息隐藏的原则 ……184
8.1.3 满足客户的需要 ……185
8.1.4 通用工具的优势 ……185
8.1.5 稳定性的价值 ……186
8.2 用计算机生成随机数……186
8.2.1 确定行为与非确定行为 ……186
8.2.2 随机数和伪随机数 ……187
8.2.3 ANSI C中生成伪随机数 ……187
8.2.4 改变随机数的范围 ……188
8.2.5 将此问题通用化 ……192
8.3 在库中保存工具 ……193
8.3.1 接口的内容 ……193
8.3.2 写random.h接口 ……195
8.3.3 random.c的实现 ……196
8.3.4 构造客户程序 ……197
8.3.5 初始化随机数发生器 ……198
8.4 评价random.h接口的设计 ……200
8.4.1 产生随机实数 ……201
8.4.2 模拟一个概率事件 ……201
8.4.3 在接口中包含头文件 ……202
8.4.4 完成随机数软件包的实现 ……202
8.5 使用随机数软件包 ……202
小结 ……205
复习题 ……206
程序设计练习 ……207
第9章 字符串和字符 ……213
9.1 枚举的原理 ……213
9.1.1 在机器内部表示枚举类型 ……214
9.1.2 将枚举类型表示为整数 ……214
9.1.3 定义新的枚举类型 ……215
9.1.4 枚举类型的操作 ……217
9.1.5 标量类型……217
9.2 字符 ……218
9.2.1 数据类型char ……218
9.2.2 ASCII代码 ……218
9.2.3 字符常量……219
9.2.4 ASCII代码方案的重要特性 ……219
9.2.5 特殊字符……220
9.2.6 字符运算……221
9.2.7 ctype.h接口 ……222

12.2 排序……311
12.2.1 对一个整数数组排序……312
12.2.2 选择排序算法……313
12.2.3 选择排序效率的评估……315
12.2.4 测试程序的运行时间……316
12.2.5 选择排序的算法分析……317
小结……318
复习题……318
程序设计练习……318
第13章 指针……323
13.1 将地址作为数据值……324
13.2 C语言的指针操作……325
13.2.1 在C语言中声明指针变量……325
13.2.2 基本的指针操作……325
13.2.3 特殊指针NULL……327
13.3 通过引用传递参数……327
13.3.1 设计函数SwapInteger……329
13.3.2 用引用调用返回多个结果……330
13.3.3 过度使用引用调用的危险……332
13.4 指针和数组……332
13.4.1 指针运算……333
13.4.2 运算符++和--的新作用……335
13.4.3 指针的自增和自减……336
13.4.4 指针和数组的关系……337
13.5 动态分配……338
13.5.1 Void*类型……339
13.5.2 动态数组……340
13.5.3 查找malloc中的错误……340
13.5.4 释放内存……341
小结……341
复习题……342
程序设计练习……344
第14章 再论字符串……347
14.1 string类型的概念表示……347
14.1.1 字符串作为数组……347
14.1.2 字符串作为指针……349
14.1.3 字符串作为抽象类型……350
14.1.4 字符串参数……350
14.1.5 字符串变量……351
14.1.6 指针和数组变量间的区别……352
14.1.7 决定字符串的表示方法……354
14.2 ANSI字符串库……354
14.2.1 strcpy函数……355
14.2.2 strncpy函数……357
14.2.3 strcat和strncat函数……358
14.2.4 strlen、strcmp和strncmp函数……359
14.2.5 strchr、strrchr和strstr函数……359
14.2.6 ANSI字符串函数的应用……360
14.3 strlib库的实现……360
14.3.1 实现转换函数……360
14.3.2 strlib分配函数的实现……363
小结……364
复习题……365
程序设计练习……365
第15章 文件……368
15.1 文本文件……368
15.2 C语言中文件的使用……369
15.2.1 声明FILE*类型的变量……370
15.2.2 打开文件……370
15.2.3 执行I/O操作……371
15.2.4 关闭文件……371
15.2.5 标准文件……371
15.3 字符I/O……371
15.3.1 文件更新……374
15.3.2 在输入文件中重新读取字符……376
15.4 面向行的I/O……377
15.5 格式化I/O……379
15.5.1 printf的三种形式……380
15.5.2 scanf函数……380
15.5.3 用scanf读入字符串……381
15.5.4 格式化I/O的一个实例……383
15.5.5 使用scanf的局限……385
小结……385
复习题……386
程序设计练习……387
第16章 记录……393
16.1 数据记录的概念……393
16.2 记录在C语言中的使用……394

第1章 概 述

[The Analytical Engine offers] a new, a vast, and a powerful language . . . for the purposes of mankind.

— Augusta Ada Byron, Lady Lovelace, *The Sketch of the Analytical Engine Invented by Charles Babbage*, 1843

学习目标

- 理解硬件和软件的区别。
- 认识问题求解是计算机科学的一个重要组成部分。
- 理解术语算法的含义。
- 理解编译器在高级程序设计语言和低级机器语言之间所起的翻译器的作用。
- 认识程序设计错误的主要类型。
- 认识软件维护的重要性和使用良好的软件工程实践方法的重要性。

1

在计算机技术高速发展的今天，很难想象在1940年计算机还不存在。如今，计算机随处可见，大家都会说，我们生活在计算机时代。

1.1 计算简史

从某种意义上说，计算的使用古已有之。许多早期数学家致力于解决重要的实际问题的计算，如观察兽群中动物的数量，计算小块土地的面积或记录商业交易等。这些活动要求人们开发新的计算技术，有时甚至要求人们发明新机器来帮助完成计算过程。例如，在亚洲使用了几千年的算盘，它是由在杆上滑动的小珠组成，这种计数装置大约在公元前2000年就出现了。

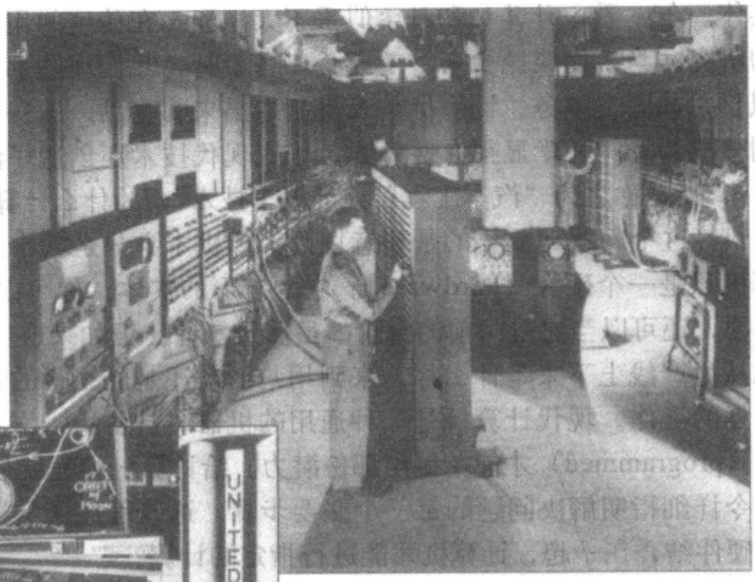
纵观计算的历史，计算的发展相对来说比较缓慢。1623年，德国科学家Wilhelm Schickard发明了第一台机械计算器，这种计算器可以自动执行简单的算术运算。尽管由于三十年战争（1618～1648）的破坏，Schickard的发明已经失传，但法国哲学家Blaise Pascal于17世纪40年代用类似的技术发明了机械加法机，其复制品现存于法国巴黎。1673年，德国数学家Gottfried Leibniz发明了一种相当复杂的装置，除进行加减运算外还可进行乘除运算。所有这些装置都是纯机械的，没有发动机和其他动力源。操作者通过将金属轮转到特定的位置来输入数字，而转动这些金属轮又可带动机器的其他部分，从而改变显示的结果。

在工业革命期间，技术的迅速进步使得用新的方法制造机械的计算机成为可能。当时，人们开始用蒸气机为工厂生产和铁路运输提供动力。在这种情况下，人们开始考虑能否用蒸气机驱动更复杂的计算机器，这种机器可在自己的动力控制下进行复杂的计算。然而在进行这项工作之前，已经有人提出问题并开始寻找答案。在这一领域中，有突出贡献的是英国数学家Charles Babbage，它是计算史上最有趣的名人之一。

Babbage一生中设计了两种不同的计算器，分别称为差分机和分析机，分别代表了当时计算器所取得的巨大进展。可惜的是，他没能完成这些项目。他所设计的用于产生数学函数表的差分机直到30年后的1854年才由一个瑞典发明家实现。分析机是Babbage一生的梦想，但直到1871年Babbage去世时这个梦想也未能实现。尽管如此，他的设计中已经包含了现代计算机的许多基本特点。最重要的是，在Babbage的构思中，分析机是一种通用的机器，可根据设计

2

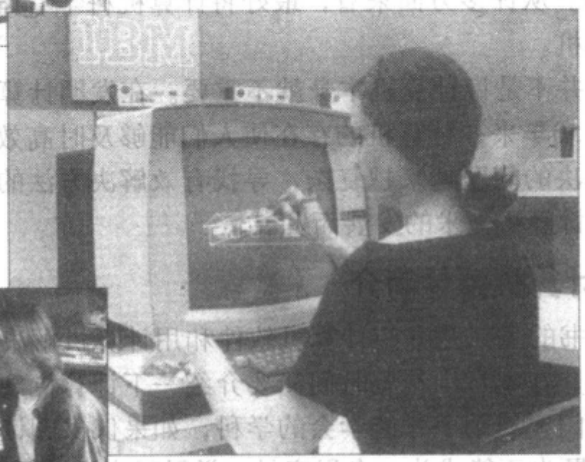
理单元装配在一个单独硅晶片上。这种技术称为**大规模集成**（large-scale integration）。由一个芯片组成的计算机处理器称为**微处理器**（microprocessor），如今已运用于大部分计算机。



通过电子化数据处理，ENIAC使美国人口普查统计数字的制表时间从12年减少为3年。但它的复杂性、高成本和大体积限制了它的吸引力和可用性。



类似于IBM 7090的计算机用穿孔卡片编程，比第一代计算机前进了一步。



集成电路使计算机的体积缩小，但功能增加。随着IBM 360的推出，商用的计算机数量大大增加。



像电话和电视一样，计算机迅速成为一种必不可少的工具，为人们提供从住所连接到世界各地数据库和在线信息中心的通路。

作为现代计算机的鼻祖，第一代和第二代计算机有着重要的历史作用，但现在看来这两种计算机既不有趣也不实用。它们就像计算机科学中的恐龙：庞大、笨拙、智力低下，很快就灭绝了。集成电路的发明者同时也是Intel公司的创始人之一Rober Noyce评论说：“与ENIAC相比，典型的现代计算机芯片比它快了二十倍，有更大的内存，可靠性提高几千倍，消耗的功率仅相当于一只灯泡而非一台机车，体积仅占其三万分之一，成本仅为其万分之一”。计算机技术已臻成熟。

例如，一条指令可能是指示计算机将两个数相加。另一条指令则将一个字符显示在计算机屏幕上。通过执行相应的一组简单指令，计算机可以完成复杂的任务。

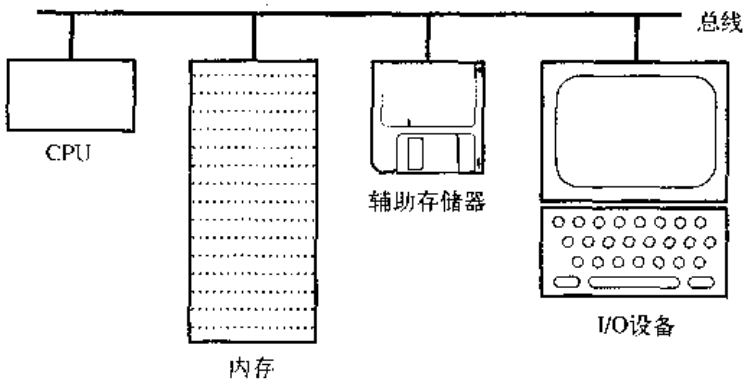


图1-1 典型的计算机组成部分

在现代计算机中，CPU由一个集成电路组成，它是一个小小的硅芯片，上面有无数微型的晶体管。这些晶体管连接在一起可以形成能够执行简单的算术和逻辑操作的电路。

1.3.2 内存

当计算机执行一个程序时，必须有一种方法来存储程序本身和运算所得的数据。总的来讲，计算机硬件中任何能够存储和检索信息的部分都是存储设备。当前运行的程序存放的存储器称为**主存储器**（primary storage），常常称为**内存**（memory）。自从1964年冯·诺依曼首次提出这个构思以来，计算机一直使用同一个存储器来存放组成程序的指令和计算过程中用到的数据。

人们将内存系统设计得十分高效，以使CPU能够以极快的速度获取它需要的信息。在如今的计算机中，内存通常构建在一个特殊的集成电路芯片上，这个集成电路芯片称作RAM，即随机访问存储器，它使程序能够在任何时刻访问任一存储单元。第11章将详细介绍内存系统的结构。

1.3.3 辅助存储器

尽管计算机运行程序时使用内存保存活动数据，但大多数内存有一个缺点，即只有在开机时能存储数据。关机后存储在其中的信息就会丢失。为使数据能永久保存，人们需要使用一种无需电力就可以保存信息的存储设备，这种设备就是**辅助存储器**（secondary storage）。

如今，计算机最常用的辅助存储器是**磁盘**（disk），它由唱片状的圆盘组成，上面涂有磁性物质以记录数据。在现代的个人计算机中，磁盘有两种形式：建造在计算机系统内部的**硬盘**（hard disk）和可移动的**软盘**（floppy disk）。当你编辑程序时，你通常在硬盘上完成你的工作（如果有可用硬盘的话）。当想要将一个程序转移到另一台计算机或出于安全的考虑要进行备份时，通常使用软盘来转移程序。

1.3.4 I/O设备

为了使计算机更加有用，必须有一些方法使其能与外界的用户沟通。计算机的输入通常由从键盘上键入的字符组成。计算机的输出结果常显示在计算机的屏幕上或打印机上。执行输入和输出操作的硬件设备合称**I/O设备**（I/O device），其中I/O表示输入/输出。

不同计算机的I/O设备有很大差别。除了标准的字母键外，不同计算机的键盘有不同的排布，甚至有些重要键的名字也不同。例如，用来表示一行结束的键在有些键盘中表示为Return，

1.5 程序设计语言和编译

用计算机解决问题包括两个概念上不同的步骤。首先，应该构造出一个算法或在解决该问题的已有算法中挑选一个。这个过程称为**算法设计**（algorithmic design）。第二步是用程序设计语言将该算法表达为程序，这个过程称为**编码**（coding）。

刚开始学习程序设计时，将算法翻译成实际的C语言的编码过程会是整个过程中较为困难的阶段。作为一个新程序员，要从简单的问题入手，就像学习任何新技术时所做的那样。简单的问题常有简单的解决方法，它的算法设计阶段不会有太大的挑战性。但由于语言及其规则是全新的，你对之并不熟悉，所以编码常常显得困难和武断。可以肯定的是，当学习了更多的有关程序设计的过程后，编码会很快变得简单起来。然而与此同时，随着要求解决的问题变得更加复杂，算法设计却变得更为困难。

本书在介绍新算法时，都会先用自然语言说明这种算法。尽管这并不像人们想象的那么精确，但在讲英语的人群中，可以用英语来表达解决方案。但如果你要向一个讲俄语的人表达你的算法，那么英语就不再是一个合适的选择。同样，在向计算机表达算法时，使用英语也是不恰当的。尽管几十年来，计算机科学家一直在研究这个问题，但让计算机理解英语、俄语或任何一种其他的人类语言还是现有技术所力不能及的。如果用人类语言表达算法，则计算机将完全不能解释算法。要想让一种算法被计算机接受，就得把它翻译成程序设计语言。 10
世界上有许多种程序设计语言，包括Fortran、BASIC、Pascal、Lisp等许多语言。本书将讲授如何使用C语言，近年来它已成为计算机业界事实上的标准。

上述包括C语言在内的程序设计语言都是被计算机科学家称为**高级语言**（higher-level language）的实例。这些语言的设计不受特性各异的计算机的影响，而是使用通用的算法概念，这种算法概念可运用于任何一个计算机系统。在内部，每个计算机系统都能理解一种低级语言，这种低级语言是它的硬件类型所决定的。例如，虽然Apple公司的Macintosh计算机和IBM的PC机都能执行用C语言这样的高级语言编写出的程序，但它们使用的基本的机器语言是不同的。

为了使用高级语言编写的程序能够在不同的计算机系统上运行，首先必须将程序翻译成运行程序的计算机特有的低级机器语言。例如，若为Macintosh机器写了一个C语言的程序，那么应该运行一个特殊的程序，该程序将C语言转换成Macintosh的专用机器语言。如果在IBM的PC机上运行该程序，则应该使用另一种翻译程序。在高级语言和机器语言之间执行这种翻译任务的程序叫做**编译器**（compiler）。

在大部分计算机系统上，运行程序之前，要先输入程序文本并将其保存在一个**文件**（file）中。文件是存储在计算机辅存里的信息集合的统称。每个文件都必须有一个文件名，通常用句点将文件名分成两部分，如`myprog.c`。建立文件时需要选择一个**根名**（root name），这是文件名中句点之前的那部分名称，它说明文件的内容。句点后的那部分名称作为**扩展名**（extension），说明文件的类型，一些扩展名有预先指定的意义。例如，扩展名.c表示文件中的程序是用C语言编写的。包含程序文本的文件称为**源文件**（source file）。

输入文件或改变文件内容的过程称为**编辑**（editing）文件。各个计算机系统之间的编辑过程差异很大，因此不可能用一种适合于各种类型的硬件的方式来描述编辑过程。当在某个计算机系统上工作时，应当学习如何建立新文件及编辑已存在的文件。这种信息可在计算机的用户手册或所使用的C编译器的说明文档中找到。

它是程序设计过程中重要的一环。

逻辑错误可能会非常难以察觉并且令人沮丧。有时程序员确信程序的算法是正确的，但随后发现它不能正确处理以前忽略了的一些情况；或者也许在程序的某个地方做了一个特殊的假定，但随后却忘记了；又或者犯了一个非常愚蠢的错误。

不用紧张，许多人都和你一样。哪怕最优秀的程序员也会有这样的经历。事实上，所有程序员都会犯逻辑错误。特别是，你还会制造一些逻辑错误。算法是一个棘手的东西，你也许常常觉得自己没能很好地掌握它。

从许多方面来讲，发现自己犯了逻辑错误是通往程序员的道路上的一个重要环节。计算机科学的先驱Maurice Wilkes在描述他在20世纪60年代早期作为一个程序员时的经历中写到：

> 在摩尔学院及其后来的一段时间，人们常常主观地认为保证程序的正确性不会有多大的困难。我清楚地记得就在那一瞬间，我明白我未来生活的很大一部分时间将花在寻找自己程序中的错误上。

优秀程序员的优秀之处并不在于他们能够避免逻辑错误，而是他们能努力将存在于完成的代码中的逻辑错误数量减到最少。当设计完一种算法并将它翻译成合乎语法的程序后，必须知道工作还未做完。几乎可以肯定，程序中的某个地方肯定会存在逻辑错误。作为一个程序员，你的工作就是不断找出并改正这种逻辑错误。应当总是怀疑自己编写的程序，并尽可能彻底地进行测试。

1.7 软件维护

软件开发的一个特殊方面是程序需要维护。事实上，对软件开发的研究表明，大多数程序在发布后，用于维护的费用占到了总费用的80%～90%。然而对于软件来说，很难想象维护到底意味着什么。这种说法听起来很奇怪。考虑一辆轿车或一座桥的情况，只有某些地方损坏时才需要维护，如一些金属部件生锈了，一个机械联动装置因过度使用而磨损了，或一些 13
部件在事故中毁坏了，这时需要进行维护。而这些情况都不会发生在软件上。代码本身不会生锈，一次次重复使用同一个程序不会在任何方面降低其可用性，偶然的错误使用当然会有危险结果，但并不会破坏程序本身，即使破坏了也可以从备份中重建程序。那么在这种情况下维护意味着什么呢？

软件需要维护主要有两个原因。首先，即使经过大量测试，并在相关领域使用多年，源代码中依然可能存在逻辑错误。其次，当出现一些不常见的情况或发生之前未预料到的情况时，之前隐藏的逻辑错误就会使程序运行失败。因此，调试是程序维护的一个基本部分。但它并非最重要的部分。更为重要的部分或许可以称为功能增强（尤其是从其占据程序总体维护费用的百分比的角度来说）。编写程序是为了使用，用程序完成顾客们需要完成的任务通常比其他方法更快更便宜。同时，程序不可能完成顾客想做的每一件事。在一个程序使用了一段时间后，顾客会期望程序能做些别的事情，或者用不同的方法做某些事情，或者用更有用的方式显示数据，或者运行得更快一些，或者有更大的容量，或者有一些更为简单但更有吸引力的特色等等。由于软件具有极大的灵活性，供应商可选择响应这些要求。不管人们是想要纠正逻辑错误还是想要增强功能，在任何情况下都需要有人查看程序，找出需要做什么，做出必要的修改，确保这些修改能正确工作，然后发布新的版本。这个过程既费时又费力，花费昂贵而且易出错。

本章重点介绍程序设计的过程所涉及的内容及其与广义计算机科学的联系。

本章的要点如下：

- 一个计算机系统的物理组件组成了硬件，这些组件是看得见、摸的着的。在硬件能够工作之前必须详细指定一系列指令，即程序，以便告诉硬件应该做什么。这样的程序叫做软件。
- 计算机科学与其说是关于计算机的科学，还不如说是用计算机解决问题的科学。
- 在计算机上解决问题的策略称为算法。成为算法的策略必须清楚明确地定义、有效、有限。
- 程序通常使用高级语言编写，然后编译器将这种高级语言翻译为特定计算机系统所使用的低级计算机语言。
- 要运行一个程序，必须先建立一个包含程序文本的源文件。编译器将源文件翻译成目标文件，它和其他目标文件链接在一起，建立一个可执行程序。
- 程序设计语言有一组语法规则以判断一个程序是否正确构建。编译器按照这些语法规则检查程序，当程序违反这些规则时就报告语法错误。
- 最严重的程序设计错误就是程序语法正确，但显示不正确的结果，甚至根本就不显示结果。由于逻辑上的错误导致程序不能正确地解决问题，这种类型的错误称为逻辑错误。找出并修正逻辑错误的过程称为调试。
- 大部分程序必须周期性地更新以纠正逻辑错误，或应应用的要求修改程序。这个过程称为软件维护，设计程序使之易于维护是软件工程的重要组成部分。
- 本书使用C程序设计语言来说明程序设计过程。尽管人们在用C语言编写程序时很少考虑软件工程的原则，但用C语言可以写出具有良好编程风格的程序。 16

复习题

1. Babbage设计的分析机为计算领域引入了什么新概念？
2. 谁是人们公认的第一名程序员？
3. 冯·诺伊曼体系结构的核心理念是什么？
4. 硬件和软件的区别是什么？
5. 传统科学涉及的是抽象理论或宇宙的本质而不是人造制品，那么，是什么抽象概念形成了计算机科学的核心？
6. 算法必须满足的三条准则是什么？
7. 算法设计和编码的区别是什么？通常情况下哪一个活动更为困难？
8. 术语高级语言的含义是什么？本书使用哪一种高级语言作为基础语言？
9. 为什么在不同厂商生产的计算机上运行程序需要使用不同的编译器？
10. 源文件和目标文件的关系是什么？程序员直接面对的是哪一种文件？
11. 语法错误和逻辑错误的区别是什么？
12. 判断题：优秀的程序员在编程中不会犯错误。
13. 判断题：编写程序最主要的开销来自程序的开发。一旦程序投入使用，程序的耗费就可以忽略不计。
14. 术语软件维护的含义是什么？
15. 为什么在编写程序时应用良好的软件工程原则很重要？ 17

第一部分

C语言程序设计基础

概要

学习C语言或任何程序设计语言在很多方面就像学习使用外语与人沟通一样。我们需要掌握一定的词汇以知道其含义，需要学习语法以便连词成句，还必须了解一些惯用语以理解人们所说的实际意义。因此，第一部分介绍C语言的词汇、语法及惯用语结构。但仅仅掌握这些还不够。就像使用任何语言一样，还必须有话可说。程序设计的精髓在于解决问题。学习如何表达固然重要，但学习如何找出解决问题的方法则是更大的挑战，因而第一部分重点讲解问题求解的策略，以使读者不仅能够像程序员一样编写程序，而且能像程序员一样去思考。

是所有C程序员共有的财富，而你也即将成为其中的一员。程序的文本见图2-1。

```
/*
 * File: hello.c
 * ------------
 * This program prints the message "Hello, world."
 * on the screen.  The program is taken from the
 * classic C reference text "The C Programming
 * Language" by Brian Kernighan and Dennis Ritchie.
 */

#include <stdio.h>
#include "genlib.h"

main()
{
    printf("Hello, world.\n");
}
```

程序注释

库包含

主程序

图2-1 hello.c

该程序本身作为一个文件存储在你所用的计算机系统的永久存储器中。文件名为hello.c，其中.c说明该文件为C语言程序。

如图2-1所示，hello.c程序分为三个部分：*程序注释*、*库包含列表*和*主程序*。尽管其结构非常简单，但它是下面几章将介绍的所有程序的典型，应将其作为C语言程序组织的范例。

2.1.1 注释

hello.c程序的第一部分只是一段英文注释，描述该程序的作用。在C语言中，**注释**(comment)是在"/*"与"*/"之间所有的文字，可以占连续的几行。在hello.c程序中，注释从第一行的/*开始，在第八行的*/后结束。

注释是写给人看的，而不是写给计算机的。他们向其他程序员传递该程序的有关信息。当C语言编译器将程序转换为可由机器执行的形式时，注释被完全忽略。

本书中的每一个程序都以一个专门的、从整体上描述程序操作过程的注释开头，这段注释称为**程序注释**(program comment)。它包括程序文件的名称和一些描述程序操作的信息。本例中的程序注释还说明了程序的来源。注释还可以描述程序中特别复杂的部分，指出可能的使用者，对如何改变程序行为提出一些建议等等。像hello.c这样简单的程序通常不需要很多的注释。但当程序越来越复杂时，读者就会发现给出合适的注释是使程序易读的最好方法之一。

2.1.2 库包含

程序的第二部分由以下两行组成：

```
#include <stdio.h>
#include "genlib.h"
```

这两行说明该程序使用了两个库。**库**(library)是一种工具的集合，这些工具由其他程序员编写，用于执行特定的功能。hello.c程序使用的库是由ANSI C提供的标准输入输出库(stdio)和专为本书设计的通用库(genlib)。本书的每个程序都会用到这两个库，也就是说，每个程序的程序注释结束后都会立即出现这两行。某些程序还会用其他的库，这些程序必须

22
|
23

但printf的作用是什么？和main一样，printf也是个函数，其名称printf代表一组操作。当想要调用这些操作时，只要使用其函数名就可以一起引用这些函数。在程序设计中，通过使用函数名来调用该函数的行为称为**调用**（calling）函数。hello.c程序中的语句

```
printf ("Hello, world.\n");
```

就是对函数printf的调用。

调用一个函数时，常常要提供一些额外的信息。例如，在C语言中，printf是一个在屏幕上显示数据的函数。但应该显示什么数据呢？这些额外信息由一组出现在函数名后的括号内的参数说明。**参数**（argument）是一个函数的调用程序提供给函数的信息。此处，printf被赋予了一个参数，它是一个列于引号中的一串字符，即**字符串**（string）

```
"hello, world.\n
```

该字符串是程序设计语言中数据的第一个例子。C语言中有许多不同类型的数据，应该特别注意如何使用数据的问题。现阶段只需将**数据**（data）看作程序所操作的信息：所有要显示的信息、用户要求的输入、作为计算结果传递的数值或计算过程的中间结果。

在本程序中，main函数中的语句通知库函数printf显示字符串中的所有字符，这些字符作为参数传入printf中。函数printf尽责地逐个显示H、e、l等等，直到屏幕上出现如下所示的完整消息。

```
Hello, world.
```

25

字符串中最后一个字符为特殊字符，称作**换行字符**，用\n表示。当函数printf到达句末的句点时，光标停留于文本的末尾的句点后。若想扩展程序使之显示更多的信息，也许会想在每个新信息开始时另起一行。**换行字符**（newline）使其成为可能，该字符在所有的计算机系统中都进行了定义。当函数printf处理换行字符时，屏幕上的光标移至下一行的起始处，就像敲击了键盘上的Return键一样（有的计算机将该键标为Enter）。在C语言中，程序必须包含换行字符以标记每一行的结尾，否则所有的输出结果将不分行地显示在一起。

2.2 两个数的加法程序

为使读者对C语言程序的工作过程有更好的了解，我们来看一个稍复杂一些的例子。图2-2所示的add2.c程序要求用户输入两个数字，将其相加并显示总和。

add2.c程序中有一些hello.c中没有的概念。首先，add2.c用到了一个新的库——simpio，即简单输入输出库。该库是本书所使用的扩展库，因此在#include行使用引号，如同genlib一样。

```
#include "simpio.h"
```

另一个值得特别注意的新特点出现在main函数的第一行：

```
int n1, n2, total;
```

这一行是*变量声明*的第一个例子。在程序中，**变量**（variable）是一些在编写程序时值未知的数据的存放处。例如，在编写两个数的加法程序时，程序员尚不知道用户要将哪两个数相加。当程序运行时，用户才会输入两个数。为了在程序中引用这些目前尚未确定的数，可创建一

3) **输出**阶段：程序在屏幕上显示计算结果。

2.2.1 输入阶段

在输入阶段，计算机必须要求用户输入两个加数并分别保存在变量n1和n2中。每个加数的输入过程包括两步。首先，程序应在屏幕上显示一个消息以使用户了解程序需要什么，这种类型的消息通常称为**提示消息**（prompt）。和向用户显示的其他消息一样，可使用printf来显示提示消息和其他消息，如下所示：

```
printf ("1st number?");
```

注意，这里在传递给printf的参数字符串后没有换行字符。省略换行字符意味着光标将停在显示的文本后面，紧接在问号之后，如下面的运行示例所示：

```
This program adds two numbers.
1st number? ▮
```

行末的光标提示用户输入一个所需的数值，而提示消息则提示应输入什么样的值。大多数情况下，用于显示输出数据的printf函数调用需要用到换行字符，用于显示输入数据提示消息的printf函数则不需要使用换行字符。

为了读取数据，程序可以这个语句：

```
n1=GetInteger();
```

该语句是一个极为重要的程序结构的实例，该结构称为**赋值语句**。在C语言中，**赋值语句**（assignment statement）将等号右边的数值存储于等号左边的变量中。在本例中，赋值语句的右边是对函数GetInteger的调用，左边是变量n1。

函数GetInteger是simpio库的一部分，用于从用户处读入整数。调用函数GetInteger时，程序等待用户用键盘输入一个完整的数值。用户输入完毕并按下回车键后，该数值作为函数GetInteger的结果返回主程序。在程序设计的术语中，称GetInteger**返回**（return）用户键入的数值。总的来说，这个赋值语句的作用就是调用GetInteger函数，让用户输入数值，并最终将GetInteger返回的数值存储于变量n1中。 28

仔细回顾以下语句的执行是很有好处的：

```
printf ("1st number?");
n1=GetInteger();
```

第一条语句仅在屏幕上显示特定的提示消息，光标停留在该行的末尾。计算机接下来执行语句

```
n1=GetInteger();
```

作为GetInteger函数调用的一部分，程序等待用户键入一个数字值，并将该数字值解释为整型值。例如，输入2时，屏幕的显示如下面的运行示例所示。（为使用户的动作更为清楚，图中用↵符号表明用户按下了回车键以表示输入行的结束，该符号实际上并不出现在屏幕上。本书中用黑体显示用户输入以便与程序产生的输出相区分。）

```
This program adds two numbers.
1st number? 2↵
```

数值2便这样存储于变量n1中了。

其中，下划线部分将由一个十进制整数代替。

要想知道显示的是什么样的整数，printf函数取调用时传来的第二个参数值，此处是变量total。它的值显示在屏幕上，换行符使光标移至下一行。最后一行加上之前已显示于屏幕上的信息就构成了程序add2.c的一个完整的运行示例：

```
This program adds two numbers.
1st number? 2↵
2nd number? 3↵
The total is 5.
```

printf函数可将任何数值作为输出的一部分显示出来。若想将一个整数作为输出的一部分，就应该在printf函数调用的第一个参数中包含格式代码%d。将要显示的实际值可以作为printf的另一个参数，按它们出现的次序排列。例如，若将add2.c的最后一行改为：

```
printf ("%d+%d=%d\n", n1, n2, total);
```

则第一个%d处将显示变量n1的值，第二个%d处将显示变量n2的值，第三个%d处将显示变量total的值。计算机屏幕上最终显示的信息为

```
This program adds two numbers.
1st number? 2↵
2nd number? 3↵
2 + 3 = 5
```

第3章中将详细讨论printf函数。 31

2.3 有关程序设计过程的观点

2.2节详细分析了add2.c程序，讲解了程序中每一个语句的特定功能。要成为一个成功的程序员，就要知道C语言中不同语句的作用和用法。有时，常常要逐句地检查程序，尤其是查找那些使程序不能正常运行的逻辑错误时。但这种详细的观察并不是检查程序的唯一方法。有时回过头从整体上检查程序会更有帮助。

再看一遍add2.c程序中的主程序，试着用一句话归纳它的作用。

```
main()
{
    int n1, n2, total;

    printf("This program adds two numbers.\n");
    printf("1st number? ");
    n1 = GetInteger();
    printf("2nd number? ");
    n2 = GetInteger();
    total = n1 + n2;
    printf("The total is %d.\n", total);
}
```

虽然说printf函数的第一次调用并不是绝对没用，但即使不看前面的说明，不知道不同类型的语句具体是怎么工作的情况下，也很可能看出这个程序的作用。它将两个数相加并显示结果。

也许更为重要的是我们可以对add2.c程序做一些修改来完成一些不同的功能。例如，可

数[⊖]的缩写。若要在程序中存储浮点型数据，就必须声明double型变量，就像在add2.c中必须声明int型变量一样。

要编写一个使用浮点型数值的完整的程序，必须先读取并显示double型数值。与整型数一样，浮点数的读取也是用simpio库中的函数完成的。要读取浮点数，需要调用GetReal函数，它的操作和本章前面2.2.1节中描述的GetInteger函数完全一样，只不过它返回的是一个double型的数值。在屏幕上显示浮点数也需要用到函数printf。但这时不是用%d而是用另一种格式代码。浮点数有多种格式代码，最简单的是用%g，它代表一般的浮点数格式。

图2-3中的编程示例说明了把add2.c程序修改为将两个浮点数相加的程序是很容易的，唯一的不同仅在于改变了变量的类型，将函数GetInteger换成了函数GetReal，将printf行中的%d换成了%g。

```
/*
 * File: add2f.c
 * -------------
 * This program reads in two floating-point numbers, adds them
 * together, and prints their sum.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double n1, n2, total;

    printf("This program adds two floating-point numbers.\n");
    printf("1st number? ");
    n1 = GetReal();
    printf("2nd number? ");
    n2 = GetReal();
    total = n1 + n2;
    printf("The total is %g\n", total);
}
```

图2-3 add2f.c

2.4.2 字符串类型的数据

尽管最初设计计算机是为了解决数值计算问题（因此计算机有时仍称作“number crunchers”），现代计算机处理的大多是文本而非数字。但由于数值计算非常简单，所以本书前几章的程序设计例子将重点介绍如何处理数值数据。实际上，熟练掌握文本数据的处理也很重要。

文本数据的最基本的元素是字符。但将字符组合起来，形成一个连续的单元时才会更有用。在程序设计中，一连串字符称为字符串，它使得在屏幕上显示消息成为可能。从hello.c开始，读者已经在程序示例中见到了一些字符串。认识到字符串也是数据，可以用操作和存储数值数据的方法来处理字符串，这一点很重要。

⊖ 该类型称为双精度浮点是因为它的精度是浮点型float的两倍。C语言开发早期普遍使用float类型，现在则多用double来表示所有的浮点数。

若用作者的名字来运行该程序，将得到如下的运行示例：

```
What is your name? Eric↵
Hello, Eric.
```

字符串对于程序设计来说非常重要，因此本书特地用几章的篇幅来讲述这部分内容。从第9章开始，本书将介绍更多的字符串的知识。

2.5 表达式

当希望程序执行计算时，应写出一个*表达式*以指定必要的操作，其形式类似于数学表达式。例如，要将add2.c程序中变量n1和n2的值相加，则正确的表达式为n1+n2。

C语言中的**表达式**（expression）由*项*和*运算符*组成。**项**（term）表示一个单独的数据值（如前一表达式中的n1和n2）。**运算符**（operator）是一个表示运算的字符或一个字符序列（如+）。在表达式中，项必须是如下选择之一：

- *常量* 程序文本中出现的任何明确的数值都称为**常量**（constant）。像0或3.1415926这样的数就是常量。
- *变量* 变量是数据的存放处，其内容在程序执行期间可以改变。
- *函数调用* 值经常通过调用其他函数来产生，这些函数可以是库函数，这些函数返回数据集给最初的表达式。在程序add2.c中，函数GetInteger用于读取每一个输入值，因此函数调用GetInterger()是一个项。函数调用将在第5章中作进一步讨论。
- *括号中的表达式* 在表达式中可以使用括号来表明操作的先后顺序，就像在数学表达式中使用括号一样。对于编译器来说，在计算开始前，括号中的表达式成为一个必须当作单元来处理的项。

当程序运行时，执行表达式中每一个特定操作的过程称为**求值**（evaluation）。当对表达式进行求值时，每一个运算符都对两边项所表示的数据值进行运算。当所有运算符求值完毕后，剩下的一个数据就是计算结果。例如，给出表达式

```
n1 + n2
```

求值过程包括读取变量n1和n2的值并将二者相加，求值结果就是两者的和。 37

2.5.1 常量

在写数学公式时，常用一些符号表示未知数，用另一些符号表示值已知的常量。例如，已知圆半径（r）求周长（C）的公式为：

$$C=2\pi r$$

为将其转换为程序语句的形式，需要使用变量来保存半径和周长。这些变量随数据的改变而改变，但2和π是值不变的常量。2是一个整型常量，π是一个实数常量，在程序中可用像3.14159这样的浮点数近似表示。由于常量是表达式的重要组成部分，因此给出每一个基本数据类型的常量值就很重要。

1) *整型常量* 要将一个整型常量作为程序的一部分或作为输入数据，只要写出组成该常量的每一个数字即可。若该整数是负数，那么在该数的前面加负号，就像数学中一样。这里

```
anExtremelyLongVariableNameWithADifferentEnding
```

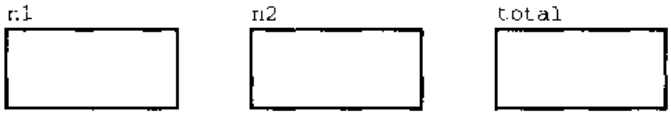
因为二者的前31个字符完全相同。为了防止这种混淆，最好避免使用多于31个字符的变量名。 39

5) 变量名应使读者易于明白其中存储的值是什么。朋友的名字、助词等虽然符合其他几条规则，但对于提高程序的可读性并无帮助。

正如在程序add2.c的讨论中指出的，在程序中使用变量之前必须明确指定每个变量的类型。这个过程称为变量的声明。变量通常在函数的开始部分声明（迄今为止，在本书中声明了变量的函数只有主函数main，但在任何函数中声明变量都是合法的）。声明本身包括变量类型名，后面是一系列作为该类型实例的变量名。例如，add2.c程序中用以下语句：

```
int  n1, n2, total;
```

声明了三个变量。这些变量的名字是n1、n2和total，它们都是int类型。这样，用我们的盒子比喻法，这个声明语句的效果就是创建了以下三个盒子，它们的名字分别为n1、n2和total。

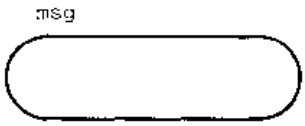


每个变量的初始值都是未定义的，所以在程序开始时不应对三个盒子中的值做出任何假设。变量n1的值可能为73，也可能是其他随机值，只有向它输入值之后才能知道里面的值是什么。

若要声明不同类型的变量，可在函数开头使用其他的声明语句。例如，若要声明string类型的变量msg，可编写以下语句：

```
string msg;
```

用盒子的比喻来说，该声明创建了一个新的名为msg的盒子。



在这个图中，用了另一种形状的盒子以强调变量msg的类型有别于n1、n2和total。例如，变量n1为int型，只能存放整型数据；变量msg为string类型，只能存放字符串数据。试图 40
将错误的数据类型存放于这些变量中，就相当于将方的物体放入圆洞中，编译器会认为这是一种错误。

2.5.3 赋值语句

正如本章前面的add2.c程序所介绍的那样，变量的值是用赋值语句获得的。C语言中的赋值语句形式如下：

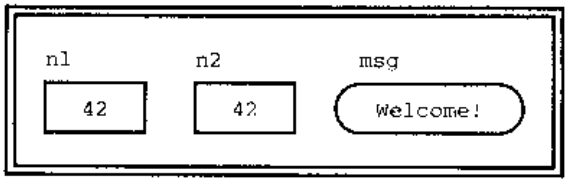
变量=表达式;

在第3章中将会讲到，上面这行语句是*程序设计范例*的一个示例。斜体字表示可以填入任何对应这些类别的内容。在写赋值语句时，等号的左边可以是任何变量名，等号右边可以是任何表达式。范例的剩余部分（本例中为等号和分号）是固定的。要写一个赋值语句，必须以一个变量名开始，然后依次是等号、表达式、分号。

本书在介绍新语句时，将使用下面的*语法框*。语法框简要介绍一种特殊语句类型以便参考。

```
n2 - n1;
```

仅改变变量n2的值，n1的值不变。

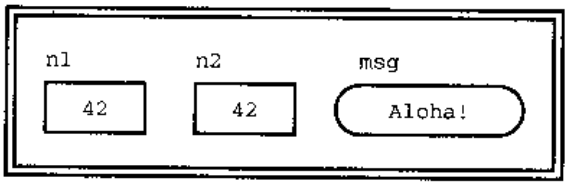


42

为变量赋予新的值将会覆盖变量中原来的值。因此，语句

```
msg = 'Aloha!';
```

将盒子图改变为如下形式：



msg中原先的值将会消失。

2.5.4 运算符和操作数

在一个表达式中，实际的运算是由连接各项的运算符表示的。最简单的运算符是数学表达式中的运算符，其中采用的是算术中的标准运算符。适用于所有数字型数据的算术运算符为：

+ 加法
- 减法（若左边无值则为负号）
* 乘法
/ 除法

每一个运算符左右两边各连接一个小的表达式，形成一个新的表达式。这些次级表达式（或称**子表达式**（subexpression））称为该运算符的**操作数**（operand）。例如，在表达式

```
x + 3
```

中，运算符“+”的操作数为子表达式x和3。操作数常为独立的项，但也可以为更复杂的表达式。例如，在表达式

```
(2 * x) + (3 * y)
```

中，“+”的操作数为子表达式(2 * x)和(3 * y)。

和传统的数学表达式一样，运算符“-”有两种形式。当“-”在两个操作数之间时，例如x-y，表示两个数相减。当其左边没有操作数时，表示负号。因此-x表示x值的相反数。此时运算符“-”称为**一元运算符**（unary operator），因为它只用于一个操作数。其余的运算符（包括表示相减的“-”）称为**二元运算符**（binary operator），因为它们适用于一对操作数。

这些新的运算符使人们可以写出比两个数相加更为有趣和实用的程序。例如，假设要编写一个程序将用英寸表示的长度转化为用厘米表示的长度，只要知道1英寸等于2.54厘米就可以了。可以修改add2f.c程序中的语句，进行适当的调整来写出其余部分。最后结果如图2-5所示。

43

```
9 % 4
```

的值为1，因为9除以4得2，余数为1。以下是%运算符的一些例子：

```
0 % 4 = 0          19 % 4   = 3
1 % 4 = 1          20 % 4   = 0
4 % 4 = 0          2001 % 4 = 1
```

/和%运算符在程序设计过程中使用非常广泛。例如，%运算符常用于检查一个数可否被另一个数整除。若要决定一个整数n能否被3整除，只要看表达式n % 3的结果是否为0即可。

当%两边的两个操作数中有一个带负号或两个均带负号时，C的行为有些混乱。不同的C语言工具处理这种情况的行为是不同的。如果完全依赖于一种行为，那么将程序放在另一台机器上运行时，结果将使人大吃一惊。为确保程序在所有机器上都能以相同的方式运行，应避免使用带负号的%操作数。

2.5.7 优先级

若一个表达式中有多个运算符，则运算的先后顺序就显得很重要了。在C语言中，可用括号括在独立的子表达式外来指定运算顺序。例如，表达式

```
(2 * x) + (3 * y)
```

45

中的括号表明在执行加法运算前，应先执行括号中的乘法运算符。但如果没有括号又会怎样呢？假设表达式为

```
2 * x + 3 * y
```

那么C编译器将如何决定每一个运算符的先后顺序呢？

和大部分的程序设计语言一样，在C语言中，优先级顺序由符合标准数学用法的一系列排序规则决定的，它们被称为**优先级法则**（rules of precedence）。对算术表达式来说，常见的规则为：

1) C编译器首先执行任何一元的负号运算符（即左边无操作数的负号）。

2) C编译器接下来执行乘法运算符（*、/、%）。若两个乘法运算符作用于同一操作数，则优先执行最左边的运算符。

3) 再接下来执行加法运算符（+和-）。同样，若两个加法运算符作用于同一操作数，则优先执行最左边的运算符。

因此，在表达式

```
2 * x + 3 * y
```

中，尽管没有括号，也应先执行乘法运算符。使用括号可使运算顺序更清楚，但本例中无需使用括号，因为想要表达的运算顺序与传统数学中既定的运算顺序相一致。若想改为首先执行加法运算符，则必须使用括号来明确表示这一事实，此时表达式应改为

```
2 * (x + 3) * y
```

优先级法则仅用于两个运算符竞争同一个操作数时。例如，在表达式

```
2 * x + 3 * y
```

中，运算符*和+竞争操作数x。优先级法则规定先执行乘法，因为乘法的优先级高于加法。同样，对于3两边的运算符，用同样的规则可以决定*应该先执行。注意，优先级法则并未明确指定应先执行哪一个乘法。这些子表达式是完全独立的，C编译器可以按任何一种顺序去执

2.5.8 优先级法则的应用

为在实际运用中说明优先级法则，先来看下面的表达式

```
8 * (7 - 6 - 5) % (4 + 3/2) - 1
```

假设自己处在计算机的位置上，将如何对此表达式进行求值呢？

第一步，先对括号内的子表达式求值，可从第一个括号（7－6＋5）开始[⊖]。为计算该表达式的值，用7减去6得1，再加上5得6。因此在对第一个子表达式求值后，式子变为

```
8 * 6 % (4 + 3/2) - 1
```

其中框表示该值是先前对子表达式求值的结果。

在第二个括号括起来的子表达式中，必须先做除法，因为乘法和除法的优先级高于加法。因此，第一步是将3除以2得1（记住，整型数据除法应舍弃余数），然后加上4得5。现在表达式变成以下形式

```
8 * 6 % 5 - 1
```

此时，C语言优先级法则指定应先做乘法和取余数操作，按照这个次序，在作减法之前，先计算6×8＝48，48除以5的余数是3。最后一步是用3减1得2，所以2为整个表达式的值。

2.5.9 类型转换

我们已经学过，在C语言中可将不同数据类型的值结合在一起。此时，C语言用**自动类型转换**（automatic type conversion）来处理这种情况。这个过程隐含在计算过程中，它将一种类型的值转换为另一种兼容的类型。例如，当用数学运算符对一个整型数和一个浮点数运算时，整型数在运算执行之前自动转换为对应的double型数。因此，若写出表达式

```
1 + 2.3
```

48

则在执行加法运算前，计算机内部先将整型数1转换为浮点数1.0。

在C语言中，当对变量赋值时也可进行自动类型转换。因此，若声明变量total为double类型，并将赋值语句写为

```
total = 0;
```

那么作为赋值操作的一部分，要先将整型数0转换为double类型。某些程序设计语言（和一些程序员）坚持将语句写成

```
total = 0.0;
```

其作用是相同的。但0和0.0在数学上的意义是不同的，因此应使用逻辑上与具体应用最适合的形式。0表示数值精确为0，因为整型数是确切值；而在数学或统计学中，0.0通常代表一个接近于0的数，它的值只精确到小数后一位。为避免混淆，本书用整型数表示确切值，即便在浮点数的情况下也是如此。

将一个double型数据赋给一个int型变量时也会激发这种自动转换，在转换过程中会舍掉小数部分。因此，若将变量n声明为int型，则赋值语句

```
n = 1.9999;
```

将会得到令人惊讶的结果，即n＝1。舍去小数部分的操作（此处和整型除法中都会出现）称作**截尾**（truncation）。

⊖ 实际上，C编译器对带括号子表达式的求值顺序是随意的，这取决于对计算机来说哪种顺序更方便，但最终结果是一样的。在编程时应注意避免导致不同求值顺序的情况。

```
quotient = num/den;
```

因为num和den均为整型数。当对两个整型数进行除法运算时，会舍去小数部分。为避免出现这种问题，必须在进行除法运算之前将至少一个值转换为double型。 50

在C语言中，可使用**强制类型转换**（type cast）来明确指出类型转换，强制类型转换是一个一元运算符，由括在括号内的目的数据类型跟上要转换的数值组成。例如，将分母的数据类型转换为double型的语句为：

```
quotient = num/(double) den;
```

既然分母现在为double类型，则按浮点数来计算除法并保留小数部分。同样的，为了转换分子的数据类型，可以将语句编写为：

```
quotient = (double) num/den;
```

这个语句有同样的作用，这是因为强制类型转换的优先级高于除法，这意味着该表达式的计算顺序和以下语句相同：

```
quotient = ((double) num)/den;
```

若强制类型转换运算符的优先级低于除法运算，则C语言将先用一个整数去除另一个整数，舍去小数部分后再将所得的整数转换为double类型，这样就得不到正确的结果。

小结

本章介绍了一些C语言程序的完整例子，使读者对C语言程序的总体结构及工作方式有所了解。主要目标是能用整体的观点看待程序设计过程本身。在本章提供的程序设计示例的基础上，现阶段应能编写包括以下一些操作的简单程序：

- 读取用户提供的数值数据。
- 在屏幕上显示文本和数据。
- 对现有数据进行算术运算得到新的结果。

本章介绍了以下一些程序设计的重要方面：

- 编写质量好的程序都包含注释，注释用自然语言解释程序在做什么。
- 许多程序使用了库提供的工具，这样程序员就不用自己重新编写这些工具了。本书中的所有程序都用到了stdio和genlib这两个库，有的程序还使用了其他库。
- 要使用库，可以在程序开始处加一个#include行以明确指定头文件。如程序示例中所示，像stdio.h这样的系统库的头文件放在尖括号中，个人库及专为本书设计的库（如genlib.h和simpio.h）放在引号中。
- 每个完整的C语言程序都有一个main函数。程序运行时，依次执行main函数中的语句。 51
- 许多程序包括以下三个阶段：输入、计算、输出。
- 接受用户输入的数据时，可以根据数据类型选择使用simpio库中的GetInteger、GetReal和GetLine函数。
- 要在计算机屏幕上显示信息和数据值，可使用stdio库中的printf函数。
- 数据值有许多不同类型，每种类型都由一个值域和一组操作构成。
- 常量用来说明程序中不变的值。
- 变量有三个属性：名字、值和类型。C程序中用到的所有变量必须要先声明，变量声明创建了变量的名字和类型。

点型常量。

a. 42	e. 20	i. 123456789
b. -17	f. 2.0	j. 0.000001
c. 2+3	g. 1,000,000	k. 1.1E+11
d. -2.3	h. 3.1415926	l. 1.1X+11

16. 将下列C语言中的浮点数常量改写成科学计数法表示形式。 53

a. 6.02252×10^{23}	c. 0.00000000529167
b. 29979250000.0	d. 3.1415926535

（每个常数都代表一个化学、物理学或数学中一个重要的近似值：a为阿佛加德罗常数；b为光速（cm/s）；c为用厘米表示的波尔半径；d为数学常数π。用科学计数法表示π并不能带来什么方便，但它是可以用科学计数法表示的，应该知道如何表示。）

17. 指出下列变量名哪一些是C语言中合法的变量名。

a. x	g. total output
b. formula1	h. aReasonablyLongVariableName
c. average_rainfall	i. 12MonthTotal
d. %correct	j. marginal-cost
e. short	k. b4hand
f. tiny	l. _stk_depth

18. 在给一个变量赋值之前，可以对它的值做出什么假设？

19. 说出下列表达式的值和类型：

a. 2 + 3	c. 19.0/5	e. 19 % 5
b. 19/5	d. 3 * 6.0	f. 2 % 7

20. 若变量k已被声明为int型变量，当程序执行赋值语句

```
k = 3.14159;
```

后，k的值是什么？若再执行下面赋值语句后，k的值是什么？

```
k = 2.71828;
```

21. 一元的负号运算符和二元的减号运算符有什么区别？

22. 应用相应的优先级法则计算下列每个表达式的值：

a. 6 + 5/4 - 3

b. 2 + 2 * (2 * 2 - 2) % 2/2

c. 10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 * 2 + 1

d. 1 + 2 + (3 + 4) * ((5 * 6 % 7 * 8) - 9) - 10

23. 在C语言中如何指定不同数据类型之间的转换？ 54

程序设计练习

1. 将本章中的hello.c程序输入计算机并运行。

2. 下列程序没有为用户提供任何注释和说明，只有几个输入提示消息：

```
#include <stdio.h>
#include "genlib.h"
#include "simpio.h"
```

$$c=\frac{5}{9}(F-32)$$

下面是该程序的运行示例：

```
Program to convert Fahrenheit to Celsius.
Fahrenheit temperature? 212↵
Celsius equivalent: 100
```

若编程时不够小心，则结果总是0。是什么逻辑错误导致这种结果？ 56

8. 在Norton Juster的童话故事*The Phantom Tollbooth*中，数学魔法师给Milo提出下面这个问题：

```
4 + 9-2 * 16 + 1/3 * 6-67 + 8 * 2-3 + 26-1/34 + 3/7 + 2-5
```

根据Milo的计算，该表达式的值总为0，数学魔法师也证实了这一点。但当我们进行计算时，只有从头开始并严格按照从左到右的顺序计算才能使结果为0。若运用C语言的优先级法则执行该表达式将得出什么结果呢？编写一个程序证明之。

9. 编写一个程序，将用千克表示的公制重量换算为用磅和盎司表示的英制重量。它们之间的换算关系为

1千克＝2.2磅　　1磅＝16盎司

10. 编写一个程序来计算四个整数的平均值。

11. 下面是一首儿歌：

As I was going to St.Ives,（在我去St.Ives的路上）

I met a man with seven wives,（我遇到了有七个老婆的男人）

Each wife had seven sacks,（每个老婆有7个麻袋）

Each sack had seven cats,（每个麻袋里有7只猫）

Each cat had seven kits:（每只猫有7只小猫）

Kits, cats, sacks, and wives,（小猫、猫、麻袋和老婆）

How many were going to St.Ives?（有多少人要去St.Ives）

最后一行其实是一个脑筋急转弯问题：只有文中的“我”去St.Ives；其他的都是去向相反的方向。假设要找出有多少小猫、大猫、麻袋和妻子是从St.Ives处过来，请编写一个C程序计算并显示结果。尝试按照问题的结构来编写程序，以便每个读到此程序的人都能理解出它计算的是什么值。 57

使用将会手到擒来。以后每当头脑中产生一个关于解决方案的某方面的大体想法时，将能很自然地把那些想法翻译成恰当的习语，构造出想要的程序。

作为程序设计习语的一个简单例子，考虑第2章中的程序add2.c。程序中的输入部分要求用户输入一个整型值。在任何情况下，add2.c都用下面的语句来完成这个任务：

1) 调用printf来显示提示消息。

2) 调用GetInteger来读入整型值。

60

这些语句构成一个程序设计习语——**读入一个整型数习语**，它的形式如下：

```
printf ("prompt string");
variable=GetInteger();
```

这两行程序是一个程序设计范例（paradigm），即展示特定语句或习语语法结构的C代码段。范例中，斜体部分需用同类成分替换。例如，在此范例中，可以在“*prompt string*”的位置填上任何提示字符串，在“*variable*”部分填上变量名。假如用“1st number”代替*prompt string*，用n1代替*variable*，就得到了add2.c中提示输入第一个数据的语句：

```
printf("1st number?");
n1=GetInteger();
```

用“2nd number?”和n2代替它们，就会得到请求输入第二个数据的语句：

```
printf("2nd number?");
n2=GetInteger();
```

*prompt string*和*variable*能用任何字符串和变量代替。通过替换，可以用同一个基本习语请求输入整型值。

3.1.1 复合赋值习语

有些C语言的习语存在主要是为一些常用操作提供方便的简化形式。其中最重要的是复合赋值操作，它在C程序中是最常见的。

在正式定义复合赋值之前，有必要知道它在什么情况下出现。假设变量balance保存某人银行账户的余额，他想往账户中存一笔钱，数额保存在变量deposit中。新的余额由表达式balance + deposit给出。于是可以写出以下赋值语句：

```
newbalance = balance + deposit;
```

然而在大多数情况下，人们不愿用一个新变量来存储结果。存钱的效果就是改变银行账户中的余额，也就是要改变变量balance中的值，使其加上新存入的数额。我们不使用上面这种把表达式的结果存入一个新的变量（如newbalance）的方法，而是把balance和deposit的值相加，并将结果重新存入到变量balance中，这种方式更可行。赋值语句可修改为：

```
balance = balance + deposit;
```

要了解这个赋值语句的作用，不能将赋值中的“=”看作数学上的等于表达式。在数学上，方程

$$x = x + y$$

61

只有当y取0时可解，除此之外，x不可能等于$x+y$。而赋值语句是一个主动的操作，它将右边表达式的值存入左边的变量中。因此，赋值语句

```
balance = balance + deposit;
```

不是断言balance等于balance+deposit。它是一个命令，使得balance的值改变为它

或

```
y=y-1;
```

++和--运算符只会在C程序中出现[⊖]。然而在第13章中将会介绍比本章所介绍的更有用和更复杂的用法。

3.2 解决规模稍大的问题

程序设计习语和范例只不过是构建程序的砖瓦。当遇到一个问题时，程序员的工作就是将这些砖瓦组织成一个连贯的能解决问题的程序。本章后面的部分将在对第2章的程序add2.c的扩展中介绍几个新的程序设计习语。

图2-2中的程序add2.c读入两个数，将它们相加并打印出它们的和。然而，把两个数相加是一项毫无挑战性的工作，如果所有问题都这么简单，计算机又有什么用武之地呢？计算机的一个最大优势就是能快速处理大量数据。在现实生活中，常见的问题不是如何将两个数相加，而是如何解决更大规模的问题。

假设要修改add2.c使之能累加10个、100个甚至1000个数，那么可以选择同样的解决方案吗？恐怕不行。对10个数来说，利用这个方案或许还能正常工作，但要声明100个变量然后使 63
用单独的语句读入每个变量的值似乎有些不可思议。如果有1000个输入值，那么利用add2.c所用的方案将会导致50多页长的重复的冗长的程序。

现在考虑一下10个数的求和问题。如何处理声明10个变量的问题？解决这样的问题使计算机程序设计有些困难，不过也正是它们使程序设计成为一件有趣的事。静下心来考虑一下这个问题，想象在没有计算机的情况下，别人依次说出10个数：7、4、6……，你将如何计算它们的和？可以将听到的数记下来，最后将它们加起来。这个方案与add2.c所用的方案类似，它的确可行，但却不是明智的和高效的。另一种可选方案是在说出数字的同时将它们加起来：7加4等于11，11加6等于17……。这样不用单独保存每一个数据，只存储当前的和。当听到最后一个数时，也就算出了10个数的和。

这个方案不用保存每个单独的数据，也就回答了如何在不声明10个变量的条件下计算10个数值的和。用这个方案可以写一个add10.c程序，此时只需使用两个变量：一个用于保存每个读入的数据，另一个用于保存当前的和。每当读入一个新的数值时，只需将它加入到保存之前所有数值和的变量中就行了。之后，又可以用该变量保存下一个数值，并按同样的方法处理。

现在可以用新的方案来编写程序了。记住，只需要声明两个变量：一个当前值，一个当前和。在程序开始添加如下的声明：

```
int value, total;
```

当输入每个数值时必须执行下面的步骤：

1) 请求用户输入一个整数值，将它存储在变量value中。

2) 将value加入到保存当前和的变量total中。

如何编写第一步的代码应该很清楚了，这正是前一节中介绍的读入一个整型数习语的典型应用，其代码为：

⊖ ++和--运算符从某种意义上说是C语言的标志性特征。没有其他任何符号能比它们更清楚地表示一个给定的程序是使用C语言而非其他语言编写的。由于这两个运算符在C中使用相当普遍，所以C语言的后继语言（采用了面向对象程序设计技术）被称为C++，因为这个名字意味着“C的后继者”。

这个for语句习语可以作为C语言中编写控制语句的具体范例。在C语言中，控制语句由两个不同的部分构成：

1) *控制行* 控制语句的第一行被称为**控制行**（control line）。它以标识语句类型的关键字开始，常常包含定义整个控制操作的其他信息。在for语句习语中，它的控制行为：

```
for (i=0; i<N; i++)
```

for语句的控制行用来控制大括号中的语句将被执行的次数。

2) *主体* 大括号中的语句构成了该控制语句的**主体**（body）。在for语句中，这些语句将按控制行指定的次数重复执行。为了方便起见，主体内的每一条语句独占一行，并且一般都比控制行多缩进四个空格，这样控制语句的控制范围就一目了然了。

控制行和主体在概念上是相互独立的。一旦写出一个控制语句的控制行，就可以在相应的主体中放入任何需要的语句。因此，for语句能用来重复执行任何操作。例如，执行语句

```
for (i=0; i<2; i++) {
    printf ("a rose is");
}
printf ("a rose.\n");
```

就得到格特鲁·斯坦因的名言

```
a rose is a rose is a rose.
```

for语句可以应用到任何语句集的特性使之成为威力巨大的编程工具。 66

3.3.2 迭代和循环

在程序设计中，重复执行一个操作的过程称为**迭代**（iteration）。迭代是许多问题解决方案的基本组成部分，特别是那些涉及大量数据的问题。一般来说，解决这类问题的程序需要对每个数据执行同样的操作。例如，add10.c对10个输入数据都重复执行读入数据和将数据加入当前和的过程。

程序员常用术语**循环**（loop）来统称在类似for语句的控制语句的执行过程中被重复执行的程序段。循环这个词来源于早期的计算，那时程序是通过穿孔纸带输入计算机的。为了重复一系列的操作，程序员将纸带的首尾相连，使纸带成为一个圈。纸带上的指令传给纸带读入机后，纸带绕一圈回来，指令又被读入，这样便达到了根据需要重复执行的目的。

当一个for循环运行时，计算机顺序执行主体内的每一条指令。当最后一条语句执行完毕时，程序返回到循环开始处并检查是否重复了指定的次数。如果是，程序从整个循环退出，接着执行for语句结尾大括号后的语句。如果还需重复，计算机将重新从循环体的第一条语句起顺序执行每一条语句。循环里所有语句的一次完全执行称为一个**周期**（cycle）。

3.3.3 下标变量

在for语句的控制行

```
for (i = 0; i < N; i++)
```

中，变量i被称为**下标变量**（index variable）。尽管可以使用任何整型变量作为下标变量，但从传统上说通常还是使用i。程序员一看到循环中的变量i，就将它看作跟踪循环的周期数的

3.3.4 初始化的重要性

for循环几乎提供了编写程序add10.c所需的一切，但还有一个重要的细节需要考虑。程序add10.c的核心部分是for循环

```
for (i=0; i<10; i++) {
    printf ("?");
    value = GetInteger();
    total += value;
}
```

它对每一个输入数据重复执行需要的操作。每当读入一个新的数值，程序将它加到变量total中，total用来保存当前所有已经读入的数值的和。

这个方案在程序运行过程中能很好地工作。例如，假设total是123，并且用户输入数值17，程序将简单地将17加入到total保存的值中，并给它一个新值140。但是total第一次的值是多少呢？第一个数值读入变量value后，通过

```
total+=value;
```

加入到total中。

众所周知，这条习语只是令计算机取value的值，将它加入total的当前值，把结果保存在total中。但total的前一次的值是什么？在循环的第一个周期，是不知道这个问题的答案的。在给变量赋值之前，它的值是未定义的。

尽管如此，它的值应该是什么是知道的。为了使程序正确地工作，在for循环的第一个周期之前，total的值必须是0，这样才能使第一个数值加入后得到正确的结果。为了保证total有正确的初始值，需要在循环开始前，明确地将它赋值为0。因此需要在程序的开始部分写下这样的语句：

```
total = 0;
```

用赋值语句来保证一个变量有正确的初始值的过程叫做**初始化**（initialization）。未初始化变量是常见的编程错误。

现在，已经介绍了完成add10.c所需的一切。完整的程序可参见图3-2。

3.3.5 读入一直到一标志习语

程序add10.c是一个说明如何使用for语句的很好的例子，但它现在的这种形式不可能满足很多用户的要求。这个程序能够准确地累加10个数值，若是要用它来累加不同个数的值时，必须要对程序进行修改，尽管这样的修改量很小，但用户真正需要的是能累加任意个数值的更通用的程序。

要想将add10.c修改为能解决累加问题的更通用的程序，有一个只需要对程序进行很小 69
修改的方法。可以在程序的开始部分请求用户输入数据个数，并将之存放在某个变量中，以此来替换for语句控制行中使用的常量10。假设n被声明为一个整型变量，程序的开始几行可以修改为

```
printf ("This program adds a list of numbers.\n");
printf ("How many numbers in the list?");
n = GetInteger();
total = 0;
for (i=0; i<n; i++)  {
...
```

有了这个新的基于标志的循环习语，就可以完成addlist.c了，它的程序参见图3-3。

```
/*
 * File: addlist.c
 * ---------------
 * This program adds a list of numbers.  The end of the
 * input is indicated by entering 0 as a sentinel value.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    int value, total;

    printf("This program adds a list of numbers.\n");
    printf("Signal end of list with a 0.\n");
    total = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetInteger();
        if (value == 0) break;
        total += value;
    }
    printf("The total is %d\n", total);
}
```

图3-3 addlist.c

在第4章中，将会抛开读入-直到-标志习语详细地介绍控制语句。在了解那些细节之前，你就也会发现这个习语是很有用的。对大多数初学者来说，抛开所有疑惑直接使用并不能完全理解的习语的确是有些困难。然而在学习程序设计的过程中，这种情况是很常见的，即使程序设计高手也不例外。实际上，程序设计高手的一大本领就是能不用理解细节而直接使用库函数或现成代码段。随着程序的日益复杂，使用那些只能从大体上理解的工具的能力是一种越来越重要的技能。

3.3.6 创造一个更实用的应用程序

累加一列数据的程序不可能卖出数百万份拷贝，使你成为软件界的百万富翁。而有谁会需要这样的加数程序呢？

当然，很多人会偶尔用到它，但他们很少会以抽象的形式思考这个问题。大多数人都会根据更特定的日常行为来思考，在日常行为中，把一列数相加是很常见的。例如，许多人每个月都会花一定的时间来结算自己的支票簿，这个行为就和加减一列数据类似。要解决这种实用的一些问题，只需要将addlist.c重新包装一下，就可以形成结算支票的程序。

要如何改动原来的程序才能成为支票结算程序呢？这在很大程度上依赖于对它的要求。一开始可以只做如下修改：

1) 修改程序开始部分的注释，以便将来读者能明白程序的作用。

2) 适当更改变量名使之更适合要解决的问题。

3) 为用户提供更明确的提示。

```
Enter the initial balance: 100↵
Enter check (-) or deposit: -50↵
Current balance = 50
Enter check (-) or deposit: -35↵
Current balance = 15
Enter check (-) or deposit: -10↵
Current balance = 5
Enter check (-) or deposit: 50↵
Current balance = 55
Enter check (-) or deposit: -25↵
Current balance = 30
Enter check (-) or deposit: 0↵
Final balance = 30
```

尽管这样的程序并没有达到你期待的通用水平，但至少它的功能对某些人来说是有用的。

3.3.7 条件执行和if语句

假设我们想给这个程序增加一些额外的特性。例如，在上面的例子中需要检测用户何时被银行退过支票。要增加这样的特性，首先要学会编写能做出判断的程序。在编写程序时，经常会遇到这样的情况：有时需要在只有满足某个条件的情况下才执行一些语句，或者根据一些测试的结果在两个可选的过程中选择一个。程序中的这类操作称为**条件执行**（conditional execution）。

C语言中表达条件执行的最简单的方法就是使用`if`语句，可以以如下两种形式使用`if`语句：

```
if (conditional-test) {
    . . . statements executed if the test is true . . .
}
```

或者

```
if (conditional-test) {
    . . . statements executed if the test is true . . .
} else {
    . . . statements executed if the test is false . . .
}
```

当解决方案只有在特定环境下才执行一系列语句时，可以使用`if`语句的第一种形式。在这种情况下，若环境不满足，则那些语句就被跳过。当解决方案需要有两种不同的过程，在满足一定条件的情况下执行其中一个，不满足条件的情况下执行另一个时，可以使用`if`语句的第二种形式。

74

范例中的*conditional-test*（条件测试）部分是一个能提出问题的特殊类型的表达式。第4章将更详细地介绍这类表达式的情况，现在只需要了解由C语言的**关系运算符**（relational operator）组成的非常简单的条件测试就够了。下面列出了C语言中定义的六个关系运算符，同时在后面的括号中给出了与它等价的数学符号。因为某些数学形式（≠、≤和≥）在标准键盘上不存在，所以C语言用两个符号的组合表示这些数学符号。

- `==` 等于(=)
- `!=` 不等于(≠)
- `>` 大于(>)

这个程序与之前版本的唯一不同之处就是下面这条加在while循环末尾的if语句：

```
if (balance<0)  {
   printf ('This check bounces. $10 fee deducted.\n");
   balance --10;
}
```

这条语句的意思是：如果余额小于0，程序将显示一条消息，并在余额中扣除一定的罚金。if语句主体的最后一条语句是

```
balance --10;
```

该语句是

```
balance - balance-10;
```

的缩略形式。

3.4 一个调试练习

75
~
76

支票结算程序中的变化是如此简单，以至于很难有第二种方案。实际上，程序员往往对程序做了一些看似很小的无伤大雅的修改，而不对修改后的程序作彻底的测试。不对代码进行测试是一个非常严重的失误。然而，更严重的失误是没有意识到所有的代码（不管代码有多简单）都需要测试。图3-5中的程序有一个很隐蔽的逻辑错误。假如只对程序进行粗略的测试，程序可以正常工作，这就使得找出逻辑错误更加困难。比如，运行下面这个程序示例，会使程序看起来能够正常工作：

```
This program helps you balance your checkbook.
Enter each check and deposit during the month.
To indicate a check, use a minus sign.
Signal the end of the month with a 0 value.
Enter the initial balance: 100↵
Enter check (-) or deposit: -50↵
Current balance = 50
Enter check (-) or deposit: -60↵
This check bounces. $10 fee deducted.
Current balance = -20
Enter check (-) or deposit: 50↵
Current balance = 30
Enter check (-) or deposit:
```

当用户输入支票金额$60时，balance的值变成负数，程序能正确判断出这个输入值超出了账户中的余额。为了让用户知道这个情况，程序输出一条消息，然后扣除$10罚金。至此，一切都很正常。

如果这样就停止测试，那么将永远不会发现程序中的逻辑错误。让我们再尝试输入另一组数据，只是将上例中最后一次存入的$50改为$10。这一次运行示例的结果如下：

```
This program helps you balance your checkbook.
Enter each check and deposit during the month.
To indicate a check, use a minus sign.
Signal the end of the month with a 0 value.
Enter the initial balance: 100↵
```

```
/*
 * File: balance3.c
 * ----------------
 * This file contains a corrected version of a program to
 * balance a checkbook, including a working bounced-check
 * feature.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

main()
{
    double entry, balance;

    printf("This program helps you balance your checkbook.\n");
    printf("Enter each check and deposit during the month.\n");
    printf("To indicate a check, use a minus sign.\n");
    printf("Signal the end of the month with a 0 value.\n");
    printf("Enter the initial balance: ");
    balance = GetReal();
    while (TRUE) {
        printf("Enter check (-) or deposit: ");
        entry = GetReal();
        if (entry == 0) break;
        balance += entry;
        if (balance < 0 && entry < 0) {
            printf("This check bounces.  $10 fee deducted.\n");
            balance -= 10;
        }
        printf("Current balance = %g\n", balance);
    }
    printf("Final balance = %g\n", balance);
}
```

图3-6 balance3.c（正确版本）

就支票结算这个程序来说，应该再多做一些测试。到目前为止，该实例中所用的所有数据都是整数。为了有更好的测试效果，需要使用包含分的数据来运行这个程序。例如，假设账户余额为$100，用户这次写了两张支票，金额分别为$49.95和$19.95。下面的运行示例展示了两次交易后的余额： 79

```
This program helps you balance your checkbook.
Enter each check and deposit during the month.
To indicate a check, use a minus sign.
Signal the end of the month with a 0 value.
Enter the initial balance: 100.00↵
Enter check (-) or deposit: -49.95↵
Current balance = 50.05
Enter check (-) or deposit: -19.95↵
Current balance = 30.1
Enter check (-) or deposit:
```

作为一个程序员，你有责任保证控制字符串中的百分号的数目与控制字符串之外传给printf的表达式数精确地匹配。但是，C编译器无法检查这条规则是否被遵守。假如有一个程序调用printf函数时，它的占位符与值的个数不符，那么继续运行它会产生不可预测的结果。 81

3.5.1 printf的格式码

printf函数的真正威力在于它能以多种格式输出数值（printf最后的字母f表示formatted（格式））。为了能准确地判断出一个值该如何显示，控制字符串中的每个百分号后面都有一个指出输出格式的关键字母。百分号和关键字母的组合称为**格式码**（format code）。在前一节的例子中，格式码是%d，它指定一个十进制整数。除了格式码%d外，程序中还出现过%g和%s，分别表示浮点数和字符串的输出。要注意，每个格式码要求相应的表达式有特定的数据类型。当使用%d时，必须保证提供的表达式值的类型为int。同样，%g需要的是浮点型值的表达式，%s需要的是字符串表达式。然而，C编译器不能检测它们的类型是否匹配，所以在实际使用时要格外小心，确保对参数使用的是适当的格式码。

常见错误 当使用printf函数时，要确保参数的个数与控制串中百分号开头的置换符个数相匹配。此外，还要确保每个参数的类型与对应的格式码是一致的。

下面列出了printf中最常见的格式码，按照应用的数据类型排列。所有可用格式码的完整列表可参阅附录A。

- %d 十进制整型。在%d格式下，数值将以标准十进制计数法的数字字符串形式显示。如果是负数，数值前会加上负号。
- %f 浮点型。在%f格式下，数值作为数字字符串显示，并在适当位置显示小数点。
- %e 指数型。在%e格式下，数值将用标准程序设计语言的科学计数法的形式显示。

 d.ddddde±xx

 这种方法对应于数学中的表示法

 $d.ddddd \times 10^{xx}$

 如果用格式码%E代替%e，那么除了用大写E来表示指数外，输出的结果是一样的。
- %g 普通型。在这种格式下，数值用%f和%e格式中较短的一个显示。如果用格式码%G代替%g，任何以科学计数法输出的数值都会用大写的E。在不能事先确定输出值有多大的情况下，%g可能是最好的输出格式。
- %s 字符串型。在%s格式下，相应的表达式也必须是字符串型，它将字符逐个显示在终端屏幕上。在该字符串中出现的百分号没有什么特殊效果。
- %% 百分号。%%并不真正指明一个格式，它只是说明%本身也是输出的一部分。 82

3.5.2 控制空格、对齐方式和精度

19世纪中叶，当查尔斯·巴贝奇（Charles Babbage）第一次设想研制自动的计算设备时，他的主要动机是让机器来创建数学函数表，以避免手工计算带来的大量错误。今天，计算机仍在创建着大量的表格数据。因此，制表和按格式组织报表仍是实际程序设计中的一个重要方面。

下面显示的这个表最大的特点之一就是数据信息是按列垂直对齐的。作为环境研究的一部分，假设你要创建一个表来展示美国每个州的总面积、森林覆盖面积以及森林覆盖率[⊖]。理想情况下，表的前几行的形式如下所示：

⊖ 数据来源：*The World Almanac and Book of Facts*, New York:Pharos Books, 1992。

里而不是50 751平方英里，它们只能给计算森林覆盖率带来很小的误差。然而，百分率的最后几位对研究来说并不重要。在这个表的第一个版本中，阿拉巴马州的森林覆盖率为66.9%，这个数实际上就足够精确了。通过指定只显示小数点后一位，就能保证表中不包含那些可能并不确切的多余的数字。[⊖]

84

printf函数通过在格式码中加入额外的格式信息来提供控制域宽、对齐方式和输出数据精度的手段。出现在百分号和关键字母之间的看似浮点数的附加信息实际是由下面所介绍的部分组成的，每一个部分都是可选的：

- 负号：它表示域中数据使用左对齐方式。若没有出现负号，则数据将右对齐。
- 数字字段宽度：它指定用于输出字段的最小字符数。假如要显示的值所占用的空间比字段宽度规定的数值小的时候，将在该域中**填充**（padded）额外的空格直到满足一定大小。如果字段宽度前没有负号，将在数据左边加额外的空格以便右对齐。如果出现了负号，那么额外的空格将加在数据右边。注意，字段宽度指定的是最小的宽度。如果数值太大，不能在指定大小的字段中显示，那么只是扩大字段宽度使之能容纳整个数值，尽管这样会打破列的对齐状态。如果没有指定字段宽度，那么数据值将使用自己需要的确切字符位数来显示，不会在两边加空格。
- 小数点及精度参数：这个格式信息的解释依赖于格式码。对%e和%f来说，精度参数指定了小数点后的位数；对%g来说，精度参数说明了最大的有效位数。对%s来说，精度参数说明字符串中最多能显示的字符个数，它避免了过长的字符串对列宽产生的不利影响。如果没有精度参数，那么printf显示整个数据。

前面给出的支票结算程序已经使用了精度参数来改进输出格式。由于支票处理的金额表示为元和分，所以按习惯指定最多显示小数点后的两位数字。用格式说明很容易达到这个目标：

```
%.2f
```

它指定输出的值要保留到小数点后的两位数字。

有了这些格式说明，便能像本节前面的范例那样用printf调用，以整齐的表格方式显示有关森林的数据信息。要让州名出现在14个字符域宽的字符串域的左端，可以使用如下的格式说明

```
%-14s
```

负号说明要左对齐；14说明域宽。如果还想保证新的字符数超过14的州名不会超出列的边界，可以用精度参数指定只显示它的前14个字符。完整的格式说明是这样的：

85

```
%-14.14s
```

使用这个格式，如果printf给定的显示字符串为“District of Columbia”，那么只有前14个字符会出现，如下所示：

```
District of Co
```

⊖ 但是，很多使用表的这种数据的人都认为，所有其中的数据都是准确的，毕竟它们是计算机计算出来的。而事实上，输出数据永远不会比输入数据准确性高，显示过多的数字只能给出精确的假象，对提高统计的准确性并无帮助。为了避免这类假象，确保不要输出超过已知正确范围的多余位数。

条件控制，它们对程序的可读性和结构的清晰性很重要。各种控制语句缩进规则将在第4章中介绍。

- 使用有意义的名字。例如，在支票结算的程序中，变量名balance清楚地告诉读者变量中包含的值是什么。假如只使用一个字母b，可能会使程序更短更容易输入，但对读者来说这个程序的可用性降低了。
- 制定变量的命名规则，使读者能够从名字中了解其功能。在本书中，变量名和数据类型名总是以小写字母开头的，例如n1，total和string等。相比之下，函数名（例如GetInteger）常常以大写字母开头。此外，若函数名是由几个单词组合而成时，就像GetInteger一样，每个单词的首字母都要大写，这样便一目了然。 87
- 在适当的情况下使用标准习语和约定。许多软件公司发布了有关程序设计风格和程序结构的内部规定，其中有些还可能与本教材中的建议相悖。俗话说入乡随俗，如果你使用的产品有内部标准，就应该遵守它，这样才能使其他程序员更容易理解。总的说来，只有在一定范围内对一些基本习惯达成共识后，程序设计才能顺利进行。
- 避免不必要的复杂性。为了程序的可读性牺牲一些程序效率常常是值得的。

我们的宗旨是让程序更容易阅读。为了达到这个目的，最好重新审校自己的程序风格，就像作家校稿一样。在进行程序设计时，最好早一些开始，完成后把它在一边放几天，然后重新把它拿出来。看看对你来说它容易读懂吗？对别人来说将来它容易维护吗？如果发现程序实际上没有意义或不易读懂，就应该花些时间来修改它。

3.6.2 设计时考虑将来的修改

编写程序时，可以采用一些能适应变更的设计以便使将来对程序的修改容易一些。因为程序员知道哪些地方比其他地方更需要修改，所以程序员常常根据经验来判断程序的那些部分应该尽可能设计得灵活些。

回想一下本章前面给出的程序balance.c。在该程序中，程序员最可能会修改的是哪一部分呢？不出意外的话，最可能修改的就是当支票退票时银行所收的罚金。程序员可以轻松地修改这个部分吗？如果现在正在写这个程序，程序员不得不仔细地检查程序，找出$10在程序的那些地方曾经出现过。随后需要做出两处修改。最明显的一处是

```
balance -- 10;
```

然而，更新前一行的printf语句也同样是重要的。

```
printf ("This check bounces. $10 fee deducted.\n");
```

考虑到以后其他人也可能会对这个程序做修改，所以希望只做一处修改，修改效果能传递到程序的所有相关部分。这样，修改程序便更加方便，并且不会因为修改程序的一部分但未修改其他相应部分而破坏程序。 88

3.6.3 #define机制

C语言中进行集中编辑修改的最好的工具就是#define结构。在最简单的情况下，#define有如下的典型形式：

```
#define symbol value
```

在这个范例中，*symbol*是一个名字，它遵循变量的命名规则；*value*是C语言中的一个常量。在#define出现过后，无论*symbol*在程序哪个部分出现，它都将被指定的常量所替代。假如在支票结算程序的开始部分写下这样的语句

```
                    BouncedCheckFee);
                balance -= BouncedCheckFee;
            }
            printf("Current balance = %.2f\n", balance);
        }
        printf("Final balance = %.2f\n", balance);
    }
```

图 3-7（续）

使用#define来设置很可能会修改的值的方法是软件工程的一个重要部分。在后面的内容中，我们将看到更多使用这种技术的例子。

小结

在第2章中，我们学习了如何编写一个简单的程序，让它读入数据，计算结果并产生输出。第3章介绍了控制语句的一些概念，以扩展你的知识。使用控制语句可以让程序解决一些更复杂的问题，如判断一定条件是否满足的测试或需要重复执行的操作。

本章鼓励读者使用控制语句解决一些力所能及的问题。每一条语句都是一个适用于特定情况的工具，之前已经展示了如何在一些简单的习语和范例中使用这些工具。第4章将更详细地介绍控制语句。 89～90

除了熟悉控制语句之外，通过示例程序balance.c的演变，读者可能已经意识到编写程序并不如看起来那么容易。特别是没有经过仔细推敲就对程序作了一些修改，或对程序没有进行彻底的测试，很容易在程序中引入一些不易察觉的逻辑错误，从而使程序得不到想要的结果。虽然从某种程度上说逻辑错误在程序设计过程中是难以避免的，但可以通过良好的程序设计规范来降低问题的严重性并节约大量时间。为了帮助读者形成自己的规范，本章给出了几个指导方针和约定来提高读者的程序设计能力。

本章中介绍的有关程序设计的重点内容如下：

- 程序中常见的操作可以表示为程序设计习语，它提供了能在各种程序设计问题中使用的简单的语句模式。
- C语言定义了几个简单的赋值运算符，使一些常见的操作更加方便。
- 只适用于两三个数据值的解决策略随着问题规模的扩大往往是不可取的。
- for语句能用来以特定次数重复执行一系列语句。
- 当在本章提供的特定的习语中使用while语句时，可以用来反复执行一系列语句直到读入一个设定的标志值为止。
- if语句用来说明一系列语句只有在一定条件满足的情况下才能被执行。而条件常常是用关系运算符比较两个数值来表示。
- 看起来似乎对程序无关紧要的修改可能会引入严重的逻辑错误，对程序应该一直保持怀疑的态度，并对它进行尽可能彻底的测试。
- printf函数提供了大量控制格式化输出的手段
- 编写的程序应该尽可能让其他程序员也容易看懂，编程时，时刻为将来的读者着想是很重要的。

复习题

1. 解释程序设计习语的概念，它在学习程序设计的过程中扮演怎样的角色？ 91

2. 编写程序，分行显示十遍“hello, world.”。

3. 以格特鲁·斯坦因的“a rose is a rose is a rose”例子作为模板，编写一个for循环，使之显示麦克白的悼词中的如下部分：

```
Tomorrow and tomorrow and tomorrow.
```

93

4. 修改程序add10.c，让它累加10个浮点型数据。

5. 编写程序，使用像下面运行示例的格式依次输出1～10的平方值。要求设计时要使上下限值便于修改。

```
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
6 squared is 36
7 squared is 49
8 squared is 64
9 squared is 81
10 squared is 100
```

6. 传说德国数学家卡尔·弗里德里奇·高斯（1777－1855）在很小的时候就崭露出他过人的数学天赋。在他小学的时候，老师让他计算1～100的和。据说高斯很快就给出了答案5050。编写一个程序解决高斯的老师提出的问题。

7. 编写一个程序，读入用户输入的5个整型值并显示其平均值，如同下面的运行示例：

```
This program averages a list of 5 integers.
? 95↵
? 100↵
? 89↵
? 91↵
? 97↵
The average is 94.4
```

注意，尽管所有的输入值都为整型，但它们的平均值可能会有小数部分。同样，设计时也要注意让使用者能方便地修改输入数据的个数。

8. 修改上题的程序，使程序在一开始时就询问用户要计算均值的数据个数，其运行界面如下图所示： 94

```
This program averages a list of integers.
How many values are there in the list? 5↵
? 95↵
? 100↵
? 89↵
? 91↵
? 97↵
The average is 94.4
```

过15个字符，则只显示其前15个字符。可以假设与会的代表不会超过10 000人，因此票数不会多于4位数字。

这里并不需要编写一个完整的程序来产生整个表，仅仅需要编写printf语句。尽管如此，仍可以试想一下如何测试这个printf语句能否达到预期效果。

13. 在第2章的练习4中编写了一个计算两年复利的程序。重写这个程序使之显示每隔N年的存款余额，其中N由用户输入。

14. 编写一个程序从用户那里读入一系列整型数据直到用户输入作为标志值的0为止。当标志值出现时，要求程序显示之前输入数据的最大值，其运行界面如下所示：

96

```
This program finds the largest integer in a list.
Enter 0 to signal the end of the list.
 ? 17↵
 ? 42↵
 ? 11↵
 ? 19↵
 ? 35↵
 ? 0↵
The largest value is 42
```

在编程之前先考虑一下这个问题，应该用什么解决方案？

如何在一列数据中找出最大值是目前为止本章中最重要的练习。一旦理解了如何解决这个基本问题（不是说如何写出相应的C语句，而是如何设计算法策略），你就可以继续学习一些程序设计的细节内容了。

97

语句。在C语言中，不论其功能如何，所有的语句都遵守以下规则：

> **简单语句规则** 一个简单语句由一个表达式及紧跟其后的分号构成。

100

因此，一个简单语句的范例如下：

```
表达式;
```

在表达式后加上分号就使它成为合法的语句形式。

尽管任何表达式后跟分号都是一个C语言的合法语句，但并不是这样组合肯定会产生有效的语句。要使一个语句有效，它必须有实际的明确的效果。以下语句

```
n1 + n2;
```

由表达式n1+n2紧跟一个分号构成，它是一个合法语句。然而，它是一个无效语句，因为它对计算结果没有进行任何操作。该语句将变量n1和n2的值相加，然后将结果丢弃[⊖]。C语言中典型的简单语句是赋值语句（包括复合形式的赋值和自增/自减操作）和能执行一些有用操作的函数调用语句，如printf。

很明显，根据简单语句规则，如下的程序行

```
printf("Hello, world.\n");
```

是合法语句。根据第2章给出的表达式的定义，函数调用也是合法表达式，因此上面语句的函数调用部分（除去分号那部分）是一个合法表达式。在其后加上分号就成为简单语句。

那么赋值语句呢？如果如下程序行

```
total = 0;
```

符合简单语句规则，那么

```
total = 0
```

本身一定是一个表达式。

在C语言中，赋值所用的等号是一个二元运算符，就像“+”或“/”。运算符“=”带两个操作数，左右两边各一个。目前，可以认为左边的操作数一定是一个变量名，尽管在第11章中会放松此限制。当赋值运算符被执行时，首先计算右边表达式的值，随后将结果存储在赋值号左边的变量中。由于用于赋值的等号是一个运算符，所以事实上

```
total = 0
```

的确是一个表达式，而

```
total = 0;
```

就是一个简单语句。

101

4.1.1 赋值的嵌套

前面对赋值的描述大家应该并不陌生，因为它和之前在第2章的2.5.3节中给出的不太正式的定义是等效的。现在有一个有趣的想法：如果一个赋值语句是一个表达式，那么表达式本身应该有值。而且，如果一个赋值表达式产生一个值，那么也一定能将这个赋值表达式嵌入到一些更复杂的表达式中去。

当赋值作为更大的表达式的一部分时，该赋值子表达式的值即为它用来赋值的值。例如，

⊖ 某些C语言编译器对这类无效语句会给出一个警告。

个独立的语句，它们都是主程序体的一部分。

为了说明一连串的语句是一个连贯的单元，可以把这些语句组装成一个**程序块**（block），用大括号把它们括起来如下所示：

```
{
    statement1
    statement2
    statement3
    . . .
    statementn
}
```

在前面章节的一些程序实例中曾经出现过程序块。每个主程序本身就是一个程序块，第3章的程序中的控制语句体也是程序块。 103

在3.6.1节中曾经讨论过，一个程序块中的语句常常根据上下文的关系有不同程度的缩进。编译器会忽略这些缩进，但其直观的效果对其他读者却是很有帮助的，因为你的程序的结构会因为这些缩进而跃然纸上。经验表明，每个新层次开始时用三到四个空格缩进会使程序的结构非常清晰，本书中每个新层次开始时将采用四个空格。适当的缩进对良好的程序设计来说是很重要的，必须要努力使自己的程序有一致的缩进风格。

程序块中唯一容易使初学者混淆的问题就是分号所扮演的角色。在C语言中，分号是简单语句语法的一部分，它的作用主要是语句*终结符*而非分隔符。虽然这条规则很简单，但对曾经使用过Pascal语言的人来说常常会有问题，因为Pascal的语法规则略有不同。具体来说，主要有以下两点：

1) 在C语言程序块的最后一个简单语句后面总会有分号。而在Pascal中常常不会出现分号，尽管大多数编译器允许将分号作为可选项。

2) 在C语言程序块的结束大括号后不会有分号。在Pascal的关键词END后，根据上下文可以有分号，也可以没有分号。

在C语言中使用分号的习惯有利于程序的维护，并且当你习惯后不会造成任何问题。

C编译器在遇到程序块时，会将整个程序块看作是一个简单语句。不管何时在习语或程序范例中看到“statement”（语句）符号，都可以用简单语句或程序块替代它。为了强调就编译器而言它们是语句，程序块有时也被称为**复合语句**（compound statement）。

4.2 控制语句

如果没有任何转向指示，C程序的语句将按照出现的次序顺序执行一次。然而，对于大多数应用来说，只有自上而下的程序顺序是不够的。现实问题的解决策略往往要求能反复执行一系列步骤或在多个动作中选择一个。能够影响其他语句的执行方式的语句叫做*控制语句*。

C语言中的控制语句分成两大类：

1) *条件*：在解决问题时，常常需要根据一些条件测试的结果，在两个或多个独立的程序执行路径中做出选择。比如，有的程序会要求当某个值为负时执行一系列操作，否则执行另一些操作。需要做出决定的那些控制语句称为**条件语句**。在C语言中，有两种条件语句的形式，即第3章介绍的`if`语句，以及本章后面将介绍的`switch`语句。 104

2) *迭代*：有时要处理的问题涉及很多数据项，则程序通常需要将某一操作重复执行若干次，或在某个条件成立的情况下反复执行某个操作。在程序设计中，这样的反复叫**迭代**。在C

```
if (x=0) . . .                    (这是错误的)
```

的结果很容易引起混淆。这条语句不会检查x的值是否为0，而是将x赋值为0，然后C语言把它理解（具体原因在这里不便说明）为测试结果为FALSE。正确的判断变量x的值是否为0的写法如下：

```
if (x==0) . . .
```

要小心避免犯这样的错误。在输入程序时稍加注意将节省大量的调试时间。

关系运算符只能用来比较**原子数据**（atomic data）值——不能再分解成更小的部分的数据。例如整型、浮点型、布尔型以及第9章中将会介绍的字符型可以组成原子数据，因为它们不能分解为更小的数据。而像字符串型（string）就不是原子数据，因为它由独立的字符组成。因 106
此可以用关系运算符比较int、double或bool型数据，但不能用这些运算符比较string型数据。在第5章中将介绍比较字符串的机制。

4.3.2 逻辑运算符

除了关系运算符外（它取任何类型的原子值并产生布尔型的结果），C语言还定义了三个运算符，它们的操作数为布尔型的值，把它们组合起来形成另一个布尔值：

- ! 逻辑非（如果后面的操作数为FALSE，则值为TRUE）。
- && 逻辑与（如果两个操作数均为TRUE，则值为TRUE）。
- || 逻辑或（如果其中一个操作数为TRUE，则值为TRUE）。

这些运算符叫做**逻辑运算符**（logical operator）。上面的运算符是按优先级递减的顺序排列的。

运算符!、&&、||可以简称为与、或、非。但要记住，有时简称并不一定能准确地表达逻辑意义。为了避免这种不准确，用更形式化、更数学化的方法来看待这些运算符是更有用的。逻辑学家用**真值表**（truth table）来定义这些运算符，它清楚地展现出布尔表达式的值如何随着操作数的值的变化而变化。例如，给定布尔值p和q，&&运算符的真值表为：

p	q	p&&q
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

该表最后一列显示，根据第一、二列布尔变量p和q的值计算出的布尔表达式p&&q的值。因此，真值表的第一行表明当p的值为FALSE，q的值为FALSE时，表达式p&&q的值为FALSE。

||操作的真值表为：

p	q	p\|\|q
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

要注意，不能把||运算符按“或”的字面意思理解为其中一个或另一个成立，而应记住它的数学意义：至少有一个成立。 107

!操作有以下简单的真值表：

常见错误 测试一个数是否在某一特定范围内，只用关系运算符把它们组合起来（就像数学中常用的方法那样）是不够的。条件的两个部分必须明确地用&&连起来，如(0 < x) && (x < 10)。

4.3.3 简化求值

C语言对||和&&运算符的解释与其他很多程序设计语言对II和&&的解释不太一样。例如在Pascal中，这些运算符（在Pascal中记作AND和OR）需要计算出条件两边的表达式的结果，尽管有时结果在计算到一半时就能确定。C语言的设计者们采用了一个不同的，对程序员来说更方便的方法。

当C程序在计算如下形式的表达式

exp1 && *exp2*

或

exp1 || *exp2*

时，子表达式总是从左到右各自被计算。一旦能确定结果就中止计算。例如，若&&表达式中的*exp1*为FALSE，则不需要计算*exp2*，因为结果能确定为FALSE。同样，在||表达式的例子中，如果第一个操作数值为TRUE，就不需要计算第二个操作数。像这种只要结果确定就停止计算的方法叫做**简化求值**（short-circuit evaluation）。

简化求值最主要的好处在于第一个条件能控制第二个条件的执行。在很多情况下，复合条件的第二部分只有在第一部分满足某个条件时才有意义。比如，要表达以下两个条件。第一，整型变量x的值非零；第二，x能整除y。由于表达式y % x只有在x不为0时才计算，所以用C语言可表达这个条件测试为：

```
(x != 0) && (y % x == 0)
```

相应的表达式在Pascal中将得不到预期的结果，因为无论何时它都要求计算出&&的两个操作数。如果x为0，尽管看起来对x有非零测试，Pascal程序还是会因为除零错误而中止。该例中 109

```
(x!=0)
```

这样的为了防止随后部分出现计算错误的条件测试叫做**监护条件**（guard）。

4.3.4 标志

bool型变量在程序设计中十分重要，它还有个特殊的名字：**标志**（flag）。例如按如下方式声明一个布尔型变量：

```
bool done;
```

变量done便成了一个标志，可以用它来记录程序中是否完成了一定的操作。可以像对其他任何变量一样给标志赋值。例如，可以写：

```
done=TRUE;
```

或

```
done=FALSE;
```

更重要的是，可以将任何值为布尔型的表达式赋值给一个布尔变量。比如，某个程序要求只要变量itemsRemaining的值为0，立即结束一些特定的操作，那么可以用如下方法给done赋予合适的值：

```
done = (itemsRemaining == 0);
```

• 该年份能被4整除但不能被100整除。

• 年份能被400整除。

假设表示年份的变量为y，以下布尔表达式将给出正确结果：

```
((y % 4 == 0) && (y % 100 != 0))  ||  ( y % 400 == 0)
```

如果考虑C语言的优先级规则，这个表达式中的括号实际上都不需要，使用括号只是为了使较长的布尔表达式更容易阅读。如果计算这个表达式的结果，并将它存储在一个叫isLeapYear的标志中，就可以在程序的其他部分通过测试标志变量得知闰年的条件是否成立。图4-1给出了计算闰年的程序。 111

```
main()
{
    int year;
    bool isLeapYear;

    printf("Program to determine whether a year is a leap year.\n");
    printf("What year? ");
    year = GetInteger();
    isLeapYear = ((year % 4 == 0) && (year % 100 != 0))
                 || (year % 400 == 0);
    if (isLeapYear) {
        printf("%d is a leap year.\n", year);
    } else {
        printf("%d is not a leap year.\n", year);
    }
}
```

图4-1 leapyear.c的主程序

4.4 if语句

C语言中表示条件执行的最简单办法就是使用if语句，它有以下两种形式：

```
If (condition) statement
If (condition) statement else statement
```

这个范例中的*condition*部分是一个布尔表达式，语句部分可以是简单语句，也可以是程序块。

当解决方案需要在满足特定条件的情况下执行一系列语句时，可以用if语句的第一种形式。如果条件不满足，构成if语句主体的那些语句将被跳过。例如，第3章balance.c的例子中的语句：

```
if (entry<0 && balance<0) {
    printf('This check bounces. $10 fee deducted.\n');
    balance-=10;
}
```

就属于这种类型：支票可能被退也可能没有被退，而程序只在支票被退的情况下才执行这些动作。

当程序必须根据测试的结果在两组动作中选择其一时，可以使用if语句的第二种形式。图4-2的程序oddeven.c说明了这种形式。该程序读入一个数，将它按奇偶分类。决定结果的条件表达式为：

```
n%2== =0
```

112

在这种情况下，如使用大括号并将if语句从一行扩展为三行，将使程序无谓地变长且不易阅读。到目前为止，这种类型的唯一例子出现在第3章的3.3.5节中：

```
if (value == Sentinel) break;
```

4.4.2 多行if语句

当if语句的主体有多条语句或有一条较长的语句使之不能与if放在一行时，这些语句将被划分为一个程序块，如语法框所示。

语法：多行if语句

```
if (condition) {
    statements;
}
```

其中：*condition*是将要测试的布尔值。

*statements*是当条件为TRUE时将要执行的一个语句块。

在这种情况下，如果条件为TRUE，将执行这些语句。如果条件为FALSE，程序不做任何动作，继续执行if语句的下一条语句。

4.4.3 if/else语句

为了避免悬空else问题，带有else子句的if语句主体将用大括号划分为两个程序块，如语法框所示。从技术上讲，只有当条件语句的主体包含不止一条语句时，才有必要在程序块前后加大括号。然而从系统地使用括号的角度上讲，加上括号会使程序产生混淆的情况大大减少，更有利于程序的维护。

语法：if/else语句

```
if (condition) {
    statements_T
} else {
    statements_F
}
```

其中：*condition*是将要测试的布尔值。

$statements_T$是当条件为TRUE时将要执行的一个程序块。

$statements_F$是当条件为FALSE时将要执行的一个程序块。

114

4.4.4 级联if语句

下面的语法框展示了当实际有两种以上情况时，if语句的一个十分重要而特殊的用法：

语法：级联if语句

```
if (condition_1) {
    statements_1
} else if (condition_2) {
    statements_2
} else if (condition_3) {                    (可以出现任意多次)
    statements_3
```

其中存储在变量value中的值即为整个?:表达式的值。

例如，可以用?:运算符将x和y中值较大的一个赋值给max，程序如下：

```
max = (x > y) ? x : y;
```

使用?:运算符的一种最常见的情况是调用printf进行输出时，输出结果可能因为某个条件而略有不同。比如编写一个程序统计某物品的数量，统计结束后将物品数量存储在变量nItems中。怎样把结果报告给用户呢？最直接的方法就是像下面这样调用printf函数：

```
printf ("%d items found.\n", nItems);
```

但假如遇到一个语言学者，当nItems的值碰巧为1时，看到这个输出可能会让他很不满意：

```
1 items found
```

因为其中复数形式的items应该是单数的。然而，可以通过在if语句中包含两个printf语句来更正这个英语语法错误：

```
if (nItems>1) {
    printf ("%d items found.\n", nItems);
} else {
    printf ("%d item found.\n", nItems);
}
```

116

遗憾的是，解决这样一个简单的问题却用了五行语句。其实，这里就可以用?:运算符来解决：

```
printf ("%d item%s found.\n", nItems, (nItems > 1) ? "s" : "");
```

输出时，如果nItems的值大于1，则在字符串item后加上一个字符串s，反之就加上一个空字符串。

另一个例子就是用?:运算符以自然语言方式输出布尔变量的值。值得注意的是，bool类型不是标准C语言的一部分，因此没有内建的输出布尔型值的机制。尽管如此，还是可以使用printf和?:来显示布尔型变量errorFlag的值：

```
printf ("errorFlag = %s\n", (errorFlag) ? "TRUE" : "FALSE");
```

在C语言中，可能会过度使用?:运算符，如果一个程序中重要的判断结构隐藏在这种嵌套的?:运算符中，后面的代码很可能就会遗漏这个判断。另一方面，如果可以使用?:代替复杂的if语句来处理一些小的细节，那它将大大简化程序的结构。

4.5 switch语句

当一个程序逻辑上要求根据特定条件做出真假判断并执行相应动作时，if语句是理想的解决方案。然而还有一些程序需要更复杂的判断结构，它们有两个以上的可选项，这些选项被划分为一系列互相排斥的情况。比如，在一种情况下，程序应该执行x；在另一种情况下，应该执行y；在第三种情况下，应该执行z，等等。在许多应用中，最适合这种情况的就是switch语句，它的语法如下框所示：

语法：switch语句

```
switch (e) {
```

时，将继续执行第二个子句。从概念的角度看，最好还是不要将这种结构看作是一个代表两种可能的case子句。

在switch语句中，default子句是可有可无的。假如没有default子句，当任何case子句都未匹配成功时，程序不做任何动作，继续执行switch结构之后的语句。为了避免程序忽略一些意外情况，在每个switch语句中都使用default子句是一种良好的程序设计习惯，除非确定已经在case语句中列举了所有的可能情况。 118

由于switch语句通常可能会很长，如果case子句本身较短，那么程序比较容易阅读。如果有足够的空间将case关键字、子句的语句和break语句放在同一行会更好。图4-4中的程序cardrank.c就是这种switch语句风格的一个例子，它是switch语句在一个玩牌程序中的应用。在这个游戏中，每一组牌用数字1～13代表，当要显示这些牌时就遇到问题了，2～10可以直接显示数字，但1、11、12和13不能直接显示数字，而应该用Ace、Jack、Queen和King来表示。程序cardrank.c用switch语句为每张牌显示适当的符号。

```
main()
{
    int n;

    printf("What is the rank of the card (1-13)? ");
    n = GetInteger();
    switch (n) {
      case  1: printf("Ace\n"); break;
      case 11: printf("Jack\n"); break;
      case 12: printf("Queen\n"); break;
      case 13: printf("King\n"); break;
      default: printf("%d\n", n); break;
    }
}
```

图4-4 cardrank.c的主程序

switch语句只能在用整型或类似整型的常量标识的case子句中选择，这大大限制了它的使用。在实际编程中，常会用字符串常量或者变量作为case子句的指示符。由于switch语句不能在这些情况下使用，所以只能用级联if语句替代。然而在条件允许的情况下，使用switch语句会使程序更具可读性、更高效。

4.6 while语句

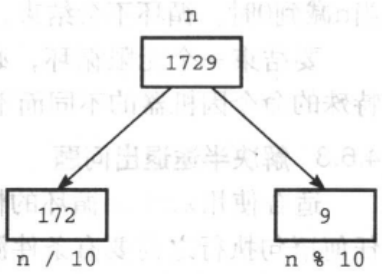
最简单的迭代结构就是while语句，它重复执行一条简单语句或程序块，直到条件表达式变为FALSE为止。while语句的范例如下面的语法框所示。与if语句一样，当循环体只由一条简单语句构成时，C编译器允许去掉加在循环体两边的括号。在本书中，为了增强程序的可读性，在所有while循环体两边都加了括号。 119

语法：while语句

```
while (condition) {
        statements
}
```

其中：*condition*是用来决定循环是否继续开始下一个周期的条件测试。
*statements*是将要重复执行的语句。

这个问题的本质在于如何将表示一个整数的各位数字拆分开来，关键是要想到用/和%这两个算术运算符可以达到目的。不难发现，任何整数的最后一位恰好是它除以10所得到的余数，写成表达式为n%10。而n/10得到的是整数n除了最后一位的其他部分。例如n的值是1729，/和%操作可以将它分成两部分：172和9，如右图所示。



因此为了求这个整数的各位数字的和，需要做的是在循环的每一个周期中将n%10的值加到变量dsum中，再将整数n除以10。下一个周期把原数的十位上的数字加到和中，如此反复直到整数的每一位都处理完毕为止。 121

但是怎样知道何时结束呢？事实上，每个周期中都将n除以10，最终将使n变成0。此时已经处理完该整数的所有数位，也就可以退出循环了。换句话说，只要n的值大于0，就应继续循环。所以，这个程序的while语句应该如下所示：

```
while (n > 0) {
        dsum += n % 10;
        n = n/10;
}
```

完整的digitsum.c程序在图4-5中给出。

```
main()
{
    int n, dsum;

    printf("This program sums the digits in an integer.\n");
    printf("Enter a positive integer: ");
    n = GetInteger();
    dsum = 0;
    while (n > 0) {
        dsum += n % 10;
        n /= 10;
    }
    printf("The sum of the digits is %d\n", dsum);
}
```

图4-5 digitsum.c的主程序

4.6.2 无限循环

当在程序中使用while循环时，一定要确保循环的控制条件最终会变为FALSE，以便能退出循环。如果while的控制行中的条件总是为TRUE，那么计算机将一个周期接着一个周期不停地执行下去，这种情况叫做**无限循环**（infinite loop）。

作为一个例子，假设在程序digitsum.c的while循环中，由于粗心将控制条件中的运算符“>”写成“>=”，如下所示：

```
while (n >= 0) {
    dsum += n % 10;                    (循环将不停地执行下去)
    n /= 10;
}
```

常见错误 仔细斟酌在while循环中使用的条件表达式，确保循环最终会结束。永远不会结束的循环称为**无限循环**。

环颠倒了这两个语句在循环中顺序，使程序较以前难读。其次，范例中要求读入数据的语句出现两次。重复的代码会带来一系列维护上的问题，因为对其中一部分的修改不会改变另外一些副本。很多经验表明，学会使用break语句来解决半途退出问题的学生比那些不会使用break语句的更能正确地编写程序[⊖]。

```
main()
{
    int value, total;

    printf("This program adds a list of numbers.\n");
    printf("Signal end of list with a 0.\n");
    total = 0;
    printf(" ? ");
    value = GetInteger();
    while (value != 0) {
        total += value;
        printf(" ? ");
        value = GetInteger();
    }
    printf("The total is %d\n", total);
}
```

图4-6 修改过后的addlist.c主程序

尽管有上述不利因素，但一些教师还是不喜欢用break来解决半途退出问题。主要是因为这样会导致C语言中break语句的滥用。防止滥用break语句的一个办法就是彻底禁用。有些人认为，这种方式太苛刻了。在本书中，仅在解决while循环的半途退出问题时使用break语句，而在其他容易引起程序结构模糊的情况下不使用break语句。

4.7 for语句

C语言中最重要的一个控制语句就是for语句。它经常在将一个操作重复执行特定次数的情况下使用。下面给出了for语句的一般形式。

语法：for语句

```
for (init; test; step) {
    statements
}
```

其中：*init*是初始化循环时计算的表达式。
*test*是一个条件测试，决定循环是否继续，就如while语句一样。
*step*是为下一个循环周期做准备的表达式。
*statements*是要被重复执行的语句。

for循环的操作是由for控制行中的三个斜体表达式*init*、*test*和*step*决定的。*init*表达式指出for循环应该如何初始化，通常用于设置下标变量的初始值。例如语句

```
for (i=0; . . .
```

⊖ 证实这个发现的最著名的研究是Elliot Soloway、Jeffrey Bonar和Kate Ehrlich合著的*Cognitive Strategies and Looping Constructs: an Empirical Study*（发表于*Communications of the ACM*, November 1983）。

```
10
 9
 8
 7
 6
 5
 4
 3
 2
 1
 0
Liftoff!
```

程序liftoff.c说明任何变量都可以作为下标变量。在这个例子中，下标变量叫做t很可能是因为它表示火箭点火升空前倒数的10秒的变量。下标变量和其他变量一样，在任何情况下都要在程序的开头对它进行声明。

表达式*init*、*test*和*step*中的每一个都是可选的，但它们中间的分号是必需的。如果*init*不出现，则将不执行任何初始化操作。如果*test*不出现，则将假设条件测试为TRUE。如果*step*不出现，则在循环的周期间将不执行任何操作。这样控制行就成了：

```
for ( ; ; )
```

它等效于

```
while (TRUE)
```

4.7.1 嵌套的for循环

当程序变得越来越复杂时，常常需要将一个for循环嵌入到另一个for循环中去。在这种情况下，内层的for循环在外层循环的每一次迭代中都将执行它的所有的周期。每个for循环都要有一个自己的下标变量以避免下标变量间的互相干扰。

图4-8中的程序timestab.c是一个嵌套for循环的一个例子。

```
/*
 * File: timestab.c
 * ----------------
 * Generates a multiplication table where each axis
 * runs from LowerLimit to UpperLimit.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * ---------
 * LowerLimit -- Starting value for the table
 * UpperLimit -- Final value for the table
 */

#define LowerLimit  1
#define UpperLimit 10
```

图4-8 程序timestab.c

可以很清楚地知道循环体中的语句将被执行10次，对应的i值从0～9。等效的while循环是：

```
i=0;
while (i<10) {
    . . . body . . .
    i++;
}
```

其中循环底部的自增操作在循环体很大时很可能会遗漏。

4.7.3 for语句中浮点型数据的使用问题

由于for循环中的*init*、*test*和*step*部分可以是任何表达式，所以没有任何明显的理由要求for循环的下标值必须是整型。在for循环中下标从0～10，每次自增2是完全可以的，代码如下：

```
for (i=0; i<=10; i+=2) . . .
```

这意味着也应该可以将x声明为double型，再将它从1.0起每次自增0.1直到2.0：

```
for (x = 1.0; x <= 2.0; x += 0.1) . . .      (这个测试可能会失败)
```

129

在有些机器上，这条语句可以得到预期结果，而在另一些机器上，可能会得不到最后一个值。例如下面这个for循环：

```
for (x = 1.0; x <= 2.0; x += 0.1) {
    printf("%.1f\n", x);                    (这个循环可能会不显示2.0)
}
```

当它在某些计算机上运行时，会产生以下输出：

```
1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
```

注意，从循环控制看来本应显示的值2.0丢失了。

问题出在浮点数并不是百分之百精确的。比如，浮点型值0.1很接近于数学上的分数1/10，但它们并不精确地相等。每当下标变量x加0.1时，误差就被累计，到最后与2.0比较来决定是否结束循环时，x的值可能就成了2.000000001或类似的不是小于或等于2.0的数。这样，for循环的条件不满足，则它在少运行一个周期的情况下就退出了。解决这个问题的最好方法就是限制只能用整型作为for循环的变量。因为整型是精确的，不会造成上述问题。

假如一定要从1.0起累加0.1到2.0，可以采用从10累加1到20，再将它除以10：

```
for (i=10; i<=20; i++) {
     x=i/10.0;
     printf ("%.1f\n", x);
}
```

复习题

1. 下面的结构在C语言中是一个合法语句吗？它有实际用处吗？

```
17;
```

2. 说明下列语句的效果，假设i、j、k声明为整型变量：

```
i = (j = 4) * (k = 16);
```

3. 用哪个简单语句可以将x和y的值设置为1.0（假设它们都被声明为double型）？

4. 结合性是什么意思？与其他运算符相比，赋值号的结合性有什么不一样？

5. 什么是程序块？它的另一个叫法——复合语句说明了程序块的什么重要特征？

6. 控制语句可以分为哪两类？

7. “两个控制语句是嵌套的”这句话是什么意思？

8. bool型数据的两个值是什么？

9. 当一个程序员试图在条件表达式中使用数学中的等号时会发生什么状况？

10. C语言对使用关系运算符来比较的数值的类型有什么限制？

11. 怎样用一个布尔表达式来测试整型变量n的值在0～9之间，并包含0和9？

12. 用自然语言描述下列条件表达式的意思： 132

```
(x != 4) || (x != 17)
```

x取什么值时这个条件为TRUE？

13. 简化求值是什么意思？

14. 假设myFlag声明为布尔型变量，下面的if语句会有什么问题？

```
if (myFlag==TRUE) . . .
```

15. 本书中使用的if语句有哪四种不同的格式？

16. 用自然语言描述switch语句的常见操作。

17. 假设在while语句的循环体中有这样一条语句，当它执行时while循环的条件值就变为FALSE。那么这个循环是将立即终止还是要完成当前周期呢？

18. 在图4-5的程序digitsum.c中，为什么要求整型值为正很重要？

19. 什么是半途退出问题？本书中提到的解决这个问题的两个策略是什么？

20. for语句控制行中出现的三个表达式的作用各是什么？

21. 当遇到下列三个情况时，你将怎样编写for语句的控制行：

a) 从1计数到100。

b) 从0开始，每次计数加7，直到成为三位数为止。

c) 从100开始，反向计数，每次减2，直到为0为止。

22. 为什么在for循环中最好避免使用浮点型变量作为下标变量？

程序设计练习

1. 为了在长途巴士上消磨时间，在美国长大的年轻人常常会反复唱下面这支歌：

99 bottles of beer on the wall.

99 bottles of beer.

You take one down, pass it around.

98 bottles of beer on the wall.

$F_5 = 5$ (2 + 3)

$F_6 = 8$ (3 + 5)

编写一个程序，顺序显示F_0到F_{15}。确保它的输出结果按以下运行示例的方式显示：

```
This program lists the Fibonacci sequence.
 F(0) =    0
 F(1) =    1
 F(2) =    1
 F(3) =    2
 F(4) =    3
 F(5) =    5
 F(6) =    8
 F(7) =   13
 F(8) =   21
 F(9) =   34
F(10) =   55
F(11) =   89
F(12) =  144
F(13) =  233
F(14) =  377
F(15) =  610
```

9. 修改上面的程序，不要指定最后一个数的下标值，而是让程序显示比10 000小的Fibonacci数列。 135

10. 编写程序在屏幕上显示下图所示图案。其中行的数量用#define定义为常量，下面是行数为8的运行示例：

```
*
**
***
****
*****
******
*******
********
```

11. 修改练习10中的程序，让它显示一个不同的三角形。在这个三角形中，每行比上一行多两个星，这个三角形如下所示：

```
       *
      ***
     *****
    *******
   *********
  ***********
 *************
***************
```

136

他要编写一段实现printf函数所需要的程序语句。由于这个工作已经做好了，其他程序员就能使用printf函数，而不需要再编写实现这个功能的语句了，甚至不需要知道实现这个功能的语句是什么。

为了更具体地考虑函数，必须先回顾一下第2章介绍的一些有关函数的基本术语。首先，**函数**（function）是由一组语句组成，并给定一个名字。执行与函数相关的一组语句的行为称为**调用**（calling）这个函数。在C语言中，函数的调用方法是写一个函数名，后面跟一组括在 138
一个圆括号中的表达式。这些表达式称为**实际参数**（argument），是调用程序传给函数的信息。在hello.c中，由于主程序将所需的数据作为函数调用的一部分，因此printf函数知道要显示什么内容。如果函数不需要从它的调用程序那里获取信息，就不需要实际参数，但是函数调用中必须要有一对空的圆括号。

一旦被调用，函数就从实际参数中获取数据，完成相应的工作，然后返回调用它的程序点。注意，调用程序所做的工作以及如何精确地返回到调用点是函数调用机制所定义的特性之一。返回到调用程序的操作称为从函数**返回**（returning）。作为返回操作的一部分，函数也能将结果返回给调用程序，如在语句

```
n1 = GetInteger();
```

中的函数GetInterger所示。在GetInteger函数完成了从用户处读取一个整数的任务后，它把这个整数作为GetInteger()调用的值返回给调用程序。这个操作被称为**返回一个值**（returning a value）。

从某种意义上讲，参数提供了函数的输入，返回值是它的输出，输出返回给调用程序。尽管在概念上是非常类似的，但严格区分输入操作（如GetInteger）和在函数作用域中使用实际参数还是很重要的。函数（如GetInteger）提供了从用户获取输入的机制。当GetInteger需要一个输入值时，终端的用户必须在键盘上输入这些值。而函数的实际参数则提供了一种使函数从它的调用程序那里获取输入值的方法，它是从程序的另一部分而不是从用户那里获取。以实际参数形式传递的数据必须由用户在程序前面的某一地方输入，并可能作为程序的某一部分计算过。也必须仔细区分输出操作（如printf）的使用和返回一个结果的技术。当使用printf时，输出显示在终端屏幕上。当函数返回一个结果时，该信息回到了调用程序，调用程序可以以任何方式使用该结果。当需要用实际参数和返回值的时候，某些新程序员会倾向于在函数内使用输入/输出操作。

常见问题 要仔细区分程序中的输入输出和函数中的实际参数和返回值的概念。输入输出允许**程序**和它的**用户**通信。参数和返回值允许**函数**和它的**调用程序**之间通信。

为了理解如何在C语言中使用函数，必须知道函数调用是一个简单的表达式，它可以出现在任何表达式可以出现的地方。而且，函数的实际参数也是一个表达式，它本身也可包含函数调用或任何合法表达式中的操作。

为了说明函数和它的实际参数是表达式，我们介绍一些math系统库中的一些标准函数。这个库包含了许多代数和三角中的标准数学函数。例如，math库包括取它实际参数的平方根的函数sqrt，以及正弦和余弦函数sin和cos。这些函数都有double类型的实际参数和返 139
回值。可以把这些函数用在简单的语句中，如

```
root3 = sqrt(3.0);
```

根。然而，C编译器不需要这个信息。它必须知道的只是sqrt取一个double类型的值，并返回一个double类型的值。函数的确切的作用是通过函数名和相关的文档告诉程序员的。

为程序员提供有用信息的另一个途径是为每一个参数提供一个描述性的名字，该名字可标识特定参数的作用（还可以包含相应的参数规格说明）。虽然参数的名字为使用该函数的程序员提供了重要的信息，但对程序没有任何实质性的影响。例如，在math.h中sin被声明为

```
double sin(double);
```

它仅指出了参数的类型。需要用这个函数的程序员可能更希望看到这个原型被写为

```
double sin(double angleInRadians);
```

以这种形式编写的原型提供了一些有用的新信息：sin函数有一个double类型的参数，该参数是以弧度表示的一个角度。在你自己的函数中，你应该为参数指定名字，并在相关的介绍函数操作的注释中使用这些名字。

如果函数没有参数，C用特殊的关键词void作为参数规格说明。例如，simpio库中的函数GetInteger没有从它的调用程序那里获取参数，返回的是一个int类型的值。这个函数的原型可以写为

```
int GetInteger(void);
```

5.3 自己编写函数

到目前为止，本章所介绍的函数都是某些程序库的一部分。库函数只对它们的正确性感兴趣，不会说明函数的全部内容。作为程序员，通常只要使用库函数，而不必知道任何内部细节。但你也有可能会使用不属于任何库的函数。在这些情况下，除了定义你自己的函数之外别无选择。

例如，你接受了一个任务，写一个程序将大多数国家使用的摄氏温度转换为美国使用的华氏温度。你可以定义一个简单的转换函数，它可以用在程序的其他部分。这个计算是相当简单的，将温度从一种计量标准转换为另一种计量标准只要使用下列公式即可。 141

$$F=\frac{9}{5}C+32$$

在C程序中增加一个新的函数由两个步骤组成：

1) 需要指定这个函数原型，它通常位于整个程序的头部，在#include行之后。

2) 在程序的稍后部分，需要提供该函数的实现，即指定一些实际的程序步骤。

原型是很短的，仅指出了函数的实际参数和返回类型。实现则较长，它要提供实现的细节。

在编写一个函数时，通常最好从编写它的原型开始。在这个例子中，你要写一个将摄氏温度转为华氏温度的函数。你可以用第2章中命名变量的一些规则，为函数选名字。这个名字要使得读程序的人容易确定函数要做什么。例如，下面的名字就不容易造成误解：

```
double CelsiusToFahrenheit(double c);
```

函数的实现是由它的原型开始，去掉结尾的分号，然后加入函数体。**函数体**（function body）是一个程序块，由一组括在一对花括号中的语句组成。程序块中的语句前面可以有变量声明，就如前面所见的那些main函数中一样。

5.3.1 return语句

如果函数返回一个结果，则函数体中至少包含一个**return语句**，它指出了要返回的值。

```
 * UpperLimit -- Final value for temperature table
 * StepSize   -- Step size between table entries
 */

#define LowerLimit   0
#define UpperLimit 100
#define StepSize     5

/* Function prototypes */

double CelsiusToFahrenheit(double c);

/* Main program */

main()
{
    int c;

    printf("Celsius to Fahrenheit table.\n");
    printf("  C    F\n");
    for (c = LowerLimit; c <= UpperLimit; c += StepSize) {
        printf("%3d  %3g\n", c, CelsiusToFahrenheit(c));
    }
}

/*
 * Function: CelsiusToFahrenheit
 * Usage: f = CelsiusToFahrenheit(c);
 * ----------------------------------
 * This function returns the Fahrenheit equivalent of the Celsius
 * temperature c.
 */
double CelsiusToFahrenheit(double c)
{
    return (9.0 / 5.0 * c + 32);
}
```

143
~
144

图 5-1 (续)

主程序由一个第4章中用来生成表的for循环组成。温度转换表中的每一行都是由语句

```
printf("%3d %3g\n", c, CelsiusToFahrenheit(c));
```

生成。这个语句调用CelsiusToFahrenheit函数计算对应于摄氏温度c的华氏温度[⊖]。结果值作为实际参数传递给printf，printf继续显示该值。

注意，图5-1的程序中有许多注释，比以前出现过的程序都多。每个函数也应该有它自己的描述性的注释，这样，程序的读者可以将每个函数作为一个单元来理解。以我的经验，为函数写的最有用的注释之一是给出一个函数是如何被使用的实例。在本书中，函数的注释都包含了一个提供这种实例的“Usage”行。

5.3.3 包含内部控制结构的函数

函数并不是都像CelsiusToFahrenheit那么简单。在许多情况下，计算一个函数需要

⊖ 在for循环中，下标变量c声明为int型，尽管CelsiusToFahrenheit被定义为取一个double的参数。正如4.7.3节指出，整型数是精确的，因此可保证程序运行正确的次数。当用参数c调用CelsiusToFahrenheit时，通过自动类型转换将参数值转换为double类型。

1) 定义一个变量product。

2) 把它初始化为1。

3) 将1～n之间的整数逐个与它相乘。

4) 将product的最终值作为函数的结果返回。

为了得到第三步中所需的每一个整数，需要一个for循环，它从1开始一直变化直到n为止。for循环需要一个下标变量，通常用i表示。因此Factorial需要声明两个变量：

```
int product, i;
```

变量product保存运行过程中的乘积值，i是下标变量。

如图5-2所示，Factorial的实现非常简单，不需要详细地解释细节问题了。

```
int Factorial(int n)
{
    int product, i;

    product = 1;
    for (i = 1; i <= n; i++) {
        product *= i;
    }
    return (product);
}
```

图5-2 Factorial函数

5.3.4 返回非数字值的函数

至今为止所介绍的函数实例返回的都是数字值，函数这个词与数学上的历史渊源使大家很容易想到创建数值函数。但在C语言中，函数可以返回任意数据类型的值。例如，要编写一个对日期进行操作的程序，那么有一个将用数值表示的月份（1～12）转换为相应的用文字表示月份名（从January到December）的string类型的函数是非常有用的。数值型的值内部工作较容易（例如，比较两个月份，看哪一个较早一些），但用文字输出月份名会使可读性更强一些。为了解决这个问题，可以定义一个函数MonthName，如图5-3所示。

```
string MonthName(int month)
{
    switch (month) {
      case  1: return ("January");
      case  2: return ("February");
      case  3: return ("March");
      case  4: return ("April");
      case  5: return ("May");
      case  6: return ("June");
      case  7: return ("July");
      case  8: return ("August");
      case  9: return ("September");
      case 10: return ("October");
      case 11: return ("November");
      case 12: return ("December");
      default: return ("Illegal month");
    }
}
```

图5-3 MonthName函数

为了使用这个函数，在程序的其他部分应该调用MonthName，然后用printf显示这个结果。例如，整数变量month、day和year的值分别为7、20、1969（阿波罗11登上月球的日子），语句

```
printf("%s %d, %d\n", MonthName(month), day, year);
```

会产生一个输出

146
～
147

如下：

```
bool IsLeapYear(int year)
{
    return (((year % 4 == 0) && (year % 100 != 0))||(year % 400 ==0));
}
```

在第4章中，已经介绍过一个确定year是否为闰年的布尔表达式。把这个表达式放入一个函数，就不需要在测试时明确地包含这个完整的计算。一旦这个函数被定义，在程序的其他地方可以用如下语句调用函数：

```
if (IsLeapYear(year)) . . .
```

5.3.6 测试字符串是否相等的谓词函数

到目前为止，你可以对字符串数据做一些简单的操作，例如用GetLine读入字符串，用printf将字符串显示在屏幕上。在第9章中，你将学到完整的字符串函数库，有了这个函数库，可以对字符串数据进行各种操作。这里将介绍这些函数中的一个。这个函数就是StringEqual，你可以用这个函数区分两个字符串是否包含完全相同的字符。

在这里介绍StringEqual函数有两个主要原因。首先，在程序设计习语中使用StringEqual使你能以更有创造性的方法使用字符串，从而写出一些更有趣的程序。其次，StringEqual是一个谓词函数，这样有助于说明谓词函数在程序设计应用中的重要性。 149

StringEqual函数的原型为：

```
bool StringEqual (string s1, string s2);
```

它指出StringEqual函数以两个字符串作为实际参数，并返回一个布尔值。如果两个字符串s1和s2中的字符完全相同，这个值为TRUE。如果两个字符串中的字符有一点点不同，StringEqual返回FALSE。下面的例子说明了这种行为：

```
StringEqual ("abc", "abc")      返回TRUE
StringEqual ("abc", "def")      返回FALSE
StringEqual ("abc", "abcd")     返回FALSE
StringEqual ("abc", "ABC")      返回FALSE
```

注意，字符串中的字符必须完全相同。如果一个字符串中多了一个字符或两个字符串中的字符稍有差别，StringEqual返回FALSE。

在需要问用户一个问题并根据回答采取某些行动时，可以使用StringEqual函数。例如，假设编写了一个与用户做游戏的程序，你要给用户提供一个重玩的机会。你可以在屏幕上提供如下适当的提示，向用户提出问题，然后调用GetLine获取响应：

```
printf("Would you like to play again?");
answer=GetLine();
```

StringEqual函数使得程序可以针对响应完成某些操作。例如，你可以写一个如下所示的主程序：

```
main()
{
    string answer;

    while (TRUE) {
        PlayOneGame();
```

```
/*
 * Function: Factorial
 * Usage: f = Factorial(n);
 * ------------------------
 * Returns the factorial of the argument n, where factorial
 * is defined as the product of all integers from 1 up to n.
 */
int Factorial(int n)
{
    int product, i;

    product = 1;
    for (i = 1; i <= n; i++) {
        product *= i;
    }
    return (product);
}
```

图 5-4（续）

如果把fact.c程序看成两个部分，该程序很有意义。主程序只是从LowerLimit到UpperLimit计数。在每一次循环中，对下标变量i的值调用Factorial函数，并显示函数结果。现在，我们通常用名字i表示用于计算for循环的循环次数的下标变量。不要感到惊奇。Factorial函数同样很直观。它比主程序复杂一点。你很容易理解函数要做什么。特别是，你可以看出n是我们要计算阶乘的数，product保存每次循环的乘积值，i是一个没有特别意义的下标变量，用来跟踪for循环的进展。这样，我们清楚了程序的每个片断的意义。

对一些新手来说，只有当把程序看成一个整体时才会糊涂。如果你读一个程序但没有理解如何看待函数，可能会出现一些有趣的问题。首先，有两个名为i的变量，一个在主程序里，一个在Factorial函数中。每个变量都作为下标变量，但这两个变量有不同的值。第二个也同样会混淆的问题是在程序的其他部分，其中两个不同的名字用来指向同一个值。在主程序中，我们要计算阶乘的数被存放在下标变量i中。在Factorial函数中，同一个值被称为n。如何搞清楚这些问题？

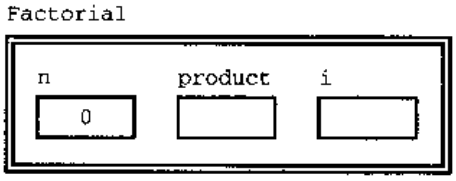
虽然把程序作为整体来看会产生混淆，但要记住当每次只看一个函数时，该函数是有意义的。如果只看main或Factorial，这些令人混淆的问题都不存在。每个函数定义只使它自己有意义。事实上，越是常见的越重要。随着程序规模的增长，没有办法将它们作为一个整体来理解。要理解大的程序，只有把它分块，每一分块足以使得它自己有意义。在fact.c程序中，求一个阶乘表的问题被分解为两个容易理解的分块。

5.4.1 参数传递

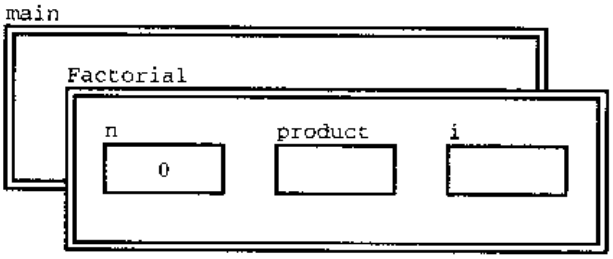
为了理解程序的两个部分是如何协同工作的，先要理解C语言本身如何处理这些混乱问题。对程序员来说，了解这个处理方法是很重要的，这使得我们可以单独考虑每个函数。C语言是如何使同一个名字有不同的值？又如何使得同一个概念值用不同的名字表示？为了理解这个问题的答案，有必要介绍一下函数调用中的实际参数值和函数中用来保存这个值的变量在语义上的区别。

当主程序调用Factorial（它本身是作为函数printf的实际参数）时，实际参数是表达式i。当这个语句被执行时，其效果相当于检查i的当前值，并把它传给函数Factorial，

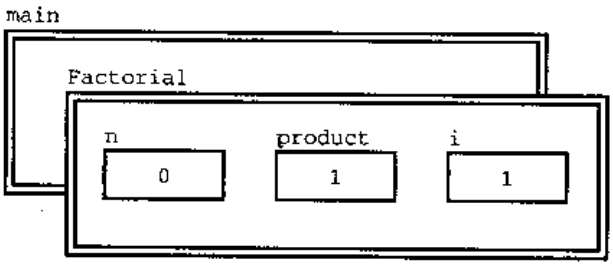
0）进行初始化。则帧的内容如右图所示。当为Factorial创建一个新帧时，main的帧没有完全去掉。该帧暂时被放在一边，直到Factorial的操作结束为止。为了在用方框图表示的概念模型上表示这种情况，最好将每个新帧图画在一张索引卡上，然后将新帧放在表示原来的帧的卡上，即覆盖在它上面。例如，当调用函数Factorial时，表示Factorial帧的索引卡放在了main帧上面。



如下图所示，整个帧集合形成了一个栈，最近的帧放在最上面，这就是栈帧这个术语的来源。一旦Factorial被激活，main的帧仍然存在，只是你看不到它的任何内容。特别要注意，名字i不再引用main函数中声明的变量，而只引用Factorial中的变量名i。

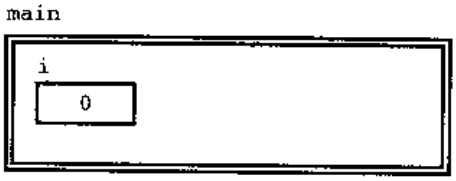


下一步是执行Factorial的函数体，方法是在当前的帧上运行函数的每一步。变量product被初始化为1，程序到达for循环。在for循环中，变量i被初始化为1，但由于该变量值已经大于n，所以for循环体没有被执行。因此，当程序到达return语句时，帧的内容变为：



155

当Factorial返回时，product的值作为函数的结果值传回调用程序。从函数返回也意味着扔掉它的帧，使main中的变量重新显现出来，如右图所示。然后，结果值1被传给printf函数。printf函数经历同样的过程。然而，操作的细节用户是不可见的，因为用户不知道printf在内部是如何工作的。最终，printf将结果值1显示在屏幕上，然后返回，之后main函数继续执行for循环的下一周期。



5.4.2 在其他函数中调用函数

函数的主要作用在于一旦一个函数被定义，不仅主程序可以使用它，而且可以作为实现其他函数的工具，使得这些函数可以用作为更复杂的工具。这些函数再被其他函数调用，以此类推，从而创建出具有任意复杂度的层次结构。

假设在你前面的桌上有n个不同物体组成的一个组。从这组物体中选出k个物体。要解决的问题是：从n个不同物体的集合中选出k个物体有多少种不同的方法？

为了使问题更加具体，假设桌上的物体是五个美元硬币：1美分、5美分、1角、25美分和

```
    printf ("Enter number of objects in the set (n)?");
    n = GetInteger ();
    printf ("Enter number to be chosen (k)?");
    k = GetInteger();
    printf ("C(%d, %d) = %d\n", n, k, Combinations (n,k));
}
```

完整的程序combine.c如图5-5所示。

```
/*
 * File: combine.c
 * ---------------
 * This program tests a function to compute the mathematical
 * combination function Combinations(n, k), which gives the
 * number of ways to choose a subset of k objects from a set
 * of n distinct objects.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Function prototypes */

int Combinations(int n, int k);
int Factorial(int n);

/* Main program */

main()
{
    int n, k;

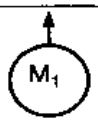
    printf("Enter number of objects in the set (n)? ");
    n = GetInteger();
    printf("Enter number to be chosen (k)? ");
    k = GetInteger();
    printf("C(%d, %d) = %d\n", n, k, Combinations(n, k));
}
/*
 * Function: Combinations
 * Usage: ways = Combinations(n, k);
 * ---------------------------------
 * Implements the Combinations function, which returns the number
 * of distinct ways of choosing k objects from a set of n objects.
 * In mathematics, this function is often written as C(n,k), but a
 * function called C is not very self-descriptive, particularly in
 * a language which has precisely the same name.
 */

int Combinations(int n, int k)
{
    return (Factorial(n) / (Factorial(k) * Factorial(n - k)));
}
```

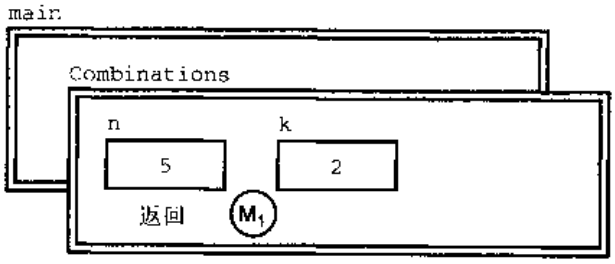
158

图5-5 combine.c

```
    printf("Enter number to be chosen (k)? ");
    k = GetInteger();
    printf("C(%d, %d) = %d\n", n, k, Combinations(n, k));
}
```

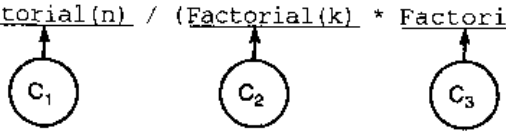


当计算机执行一个新的函数调用时，它记录了一旦调用完成后调用程序应该从哪里继续执行。执行的继续点被称为**返回地址**（return address），在这些图中用一个圆形标记指出。为了记住程序执行的位置，应该在帧图上记录调用点，如： 160

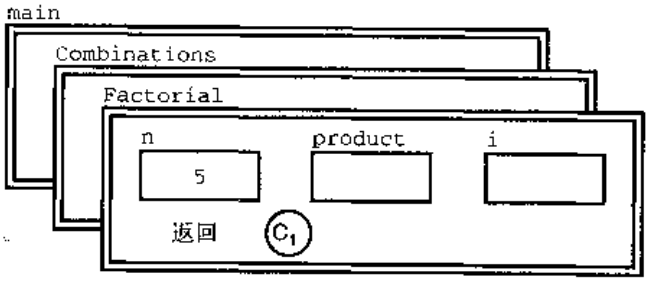


一旦创建新的帧，程序开始执行Combinations函数的函数体，它将用标记记录每个Factorial函数的调用：

```
int Combinations(int n, int k)
{
    return (Factorial(n) / (Factorial(k) * Factorial(n - k)));
}
```



为了执行这个语句，计算机必须做出三个调用函数Factorial的标记。按照ANSAI C制定的标准，编译器可以以任何次序做出这些调用。在这种情况下，假设选择按从左到右的顺序计算。第一个调用请求计算机计算Factorial(n)，这将创建如下的新帧：



Factorial函数按前面已介绍过的过程运行，并返回值120到标记C_1指出的调用点。Factorial函数的帧就消失了，回到Combinations的帧中，继续下一步计算。可以用返回并将调用结果填在调用的位置的方法来说明当前的状态，如下所示： 161

```
return (120 / (Factorial(k) * Factorial(n - k)))
```



图中120外面的方框指出这个值不是程序的一个部分，而是一些以前计算的结果。

```
C(5, 2) = 10
```

这是该程序的最后一个操作。

这个深入剖析内部细节的练习是为了帮助你理解C语言中的函数调用机制。你将发现利用这种方法将自己的程序检查一遍或两遍是很有帮助的，但不要把它当成一种习惯。相反，你应该学会用非形式化的方法考虑函数，尝试从直观上理解函数是如何工作的。当程序调用一个函数时，函数执行它自己的操作，然后程序从调用点继续执行。如果函数返回一个结果，调用程序在以后的计算中可以自由应用此结果。作为一个程序员，你应该找到一个点，在此你可以舒服地考虑这个过程，而不需要为细节问题而烦恼。毕竟，计算机会为你考虑这些问题。

5.5 过程

到目前为止，本章中的函数都为它们的调用程序返回某种类型的值。在程序设计中，有时调用一个函数不是为了得到它的返回值，而是要它产生某些作用，这种用法是相当普遍的。 163
例如，当你在一个语句中用以下方法调用printf函数时，

```
printf("Hello world.\n");
```

你并不希望让printf返回某些值。你所关心的是在屏幕上显示某些信息。

不返回值的而是起到某些作用的函数被称为过程（procedure）。许多程序设计语言，包括Pascal和Fortran，提供了完全不同的机制定义函数和过程。在这些语言中，函数和过程是两个不同的概念实体。在C语言中，这两个概念是合在一起的，大多数有关C语言的书都用术语函数表示函数和过程。在本书中，我们基本遵循这个习惯。但认识到某些函数是不返回结果的，这很重要。因此，当强调某个函数没有返回值时，本书会使用过程这个词。

在C语言中，通过在函数原型中返回类型的地方用关键词void来表示过程。例如，原型

```
void GiveInstructions (void);
```

声明一个没有参数和返回值的函数。对任何函数来讲，原型出现在程序的开始部分；而GiveInstructions的实现则出现在程序后面的部分中。这个过程的一般的实现如下所示：

```
void GiveInstructions(void)
{
    printf ("This program performs some important calculation\n");
    printf ("for the user. This function, or one very much\n");
    printf ("like it, can be used to give the user whatever\n");
    printf ("instructions are required to use the program,\n");
    printf ("such as the format for input data and the like.\n");
}
```

与大多数过程一样，GiveInstructions没有包括return语句，而是直接“跌到了终点”，C语言解释这种情况为过程的执行完成了。然而，过程也可以用return语句强制立即从过程返回。当return语句用在过程中时，单词return后面没有表达式。

一旦定义GiveInstructions过程，主程序为了在屏幕上依次显示这些指令时，就要调用这个过程。因此可假定主程序从下列语句开始：

解的特殊情况或限制下结束时更是如此。逐步精化的原则指出：一旦你有了某个程序的概要描述，你就应该在此结束，并把它写下来，调用过程或函数表示那些还要继续细化的程序。如果采用这种方法，主程序可以写成：

```
main()
{
    int year;

    GiveInstructions();
    Year =GetYearFromUser();
    PrintCalendar(year);
}
```

第一个语句调用GiveInstructions过程，该过程的实现与本章前面的5.5节中出现的GiveInstructions过程非常类似。第二个语句调用函数GetYearFromUser，该函数可用来处理读入年份的过程。GetYearFromUser的实现可以简单地表示为显示一个提示符，并调用GetInteger，但也可以包括一些其他的操作，如检查用户输入的年份是否合法。主程序的最后一行调用函数PrintCalendar，给它传递所需的年份作为参数。最后一个函数的实现将涉及解决整个问题的绝大部分工作，但这个问题现在已经变得稍微简单一些了，因为主程序已经考虑了所有与用户的交互。

在main实现所示的详细级别上，程序已经具有完整的意义。只要GiveInstructions、GetYearFromUser和PrintCalendar完成它们的工作，程序就应该正常工作。当然，你还没有编写这些函数的实现，因此现在就依赖于他们似乎太早。然而，上述工作是成功分解的关键。只要沿着这条路进行下去，你会发明新的过程和函数来解决这些任务中的新的有用的片段，然后，根据这些片段实现解决方案层次结构中的每一层。 166

5.6.2 实现PrintCalendar

写完了主程序之后，还要编写三个函数。虽然你可以从编写任何一个函数开始，但我们直接进入问题的中心，实现PrintCalendar。从main程序的调用判断，PrintCalendar的原型是

```
void PrintCalendar (int year);
```

它的实现怎么写？

在抽象级别考虑PrintCalendar函数，考虑它要做些什么。它通过打印12个单独的月历完成日历的显示。利用这些想法足以编写一个适合这个分解层次的该函数的实现。PrintCalendar的函数体是一个简单的循环，它调用另外一个函数显示某一个月的月历，然后打印一个换行符使得该月与下一个月之间有一个空行：

```
void  PrintCalendar(int year)
{
    int month;

    for (month = 1; month <= 12; month++) {
        PrintCalendarMonth(month, year);
        Printf ("\n");
    }
}
```

PrintCalendarMonth函数有两个参数，月份和年份，因为这两个信息它都需要。参数month

用在这里就不太合适，weekday最终会成为7、8、9等值，这些值没有对应的星期几。通过除7取余可以保证结果值总是在0～6之间。当采用取余数操作将计算的结果限制在一个较小的周期性的范围内时，使用的是数学上称为**取模运算**（modular arithmetic）的过程。取模运算在程序设计中非常有用，在本书中还可以看到一些其他实例。

如果你保存了星期几的记录，那么在PrintCalendarMonth函数中写一个主循环并不困难。下面的代码就能完成这些工作：

```
for (day = 1; day <= nDays; day++) {
    printf("%2d", day);
    if (weekday == Saturday) printf("\n");
    weekday = (weekday + 1) % 7;
}
```

该循环显示了每一个数，记录了它是星期几，在每个星期六之后换行。日历的最后一行必须以换行结束，因此循环后必须紧跟以下的语句：

```
if (weekday != Sunday) printf ("\n");
```

这样就保证即使这个星期不是在星期日结束，最后一行后面也有换行符。

至此，还剩下三个任务：

1) 计算出该月的天数。

2) 确定该月的第一天是星期几。

3) 缩排月历的第一行，使得第一天出现在正确的位置。

逐步精化的策略建议不要在分解的这个层次上解决这三个问题。事实上，可以将这三个操作转化为函数调用，这些函数可以以后再实现。应用这个策略，可以写一个完整的PrintCalendarMonth的实现：

```
void PrintCalendarMonth(int month, int year)
{
    int weekday, nDays, day;

    printf("    %s %d\n", MonthName(month), year);
    printf(" Su Mo Tu We Th Fr Sa\n");
    nDays = MonthDays(month, year);
    weekday = FirstDayOfMonth(month, year);
    IndentFirstLine(weekday);
    for (day = 1; day <= nDays; day++) {
        printf(" %2d", day);
        if (weekday == Saturday) printf("\n");
        weekday = (weekday + 1) % 7;
    }
    if (weekday != Sunday) printf("\n");
}
```

169

用逐步精化的方法实现PrintCalendarMonth意味着将其中用到的三个函数（MonthDays、FirstDayOfMonth和IndentFirstLine）的实现将放到以后再写。当你回过头来填补这些缺少的部分时，你可以按任意次序来实现这些函数，但在你执行程序前必须编写完成所有的函数。

最容易实现的函数是这组函数中的最后一个：IndentFirstLine。这个函数从FirstDayOfMonth中获取一个星期中的第几天，保证月历的第一行前有足够的空格，使第一天出现在正确的位置。如果这个月从星期天开始，月历将在第一行的起始位置开始。如果从星期一开始，程序将打印出表示一天所需的空格来表示缺少星期天。因为每个月历上的项目占用3

```
        weekday = (weekday + MonthDays(i, year)) % 7;
    }
    return (weekday);
}
```

5.6.4 完成最后的片段

完成这个程序只需要填充一些细节。例如，需要实现函数GiveInstructions和GetYearFromUser。FirstDayOfMonth的定义要求处理的年份不能早于1900年，因此最好能在这些函数中做一些限制，强迫满足这个要求。例如，下面的GetYearFromUser的实现检查输入的年份是否满足这个条件，如果不满足，则给用户一个重新输入的机会。

```
int GetYearFromUser(void)
{
    int year;

    while (TRUE) {
        printf("Which year? ");
        year = GetInteger();
        if (year >= 1900) return (year);
        printf("The year must be at least 1900.\n");
    }
}
```

171

剩下的工作还有需要从本章前面拷贝IsLeapYear和MonthName函数的实现，确保包括所有的函数原型、写注释、最终编译和测试程序。完整的实现如图5-6所示。

```
/*
 * File: calendar.c
 * ----------------
 * This program is used to generate a calendar for a year
 * entered by the user.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants:
 * ----------
 * Days of the week are represented by the integers 0-6.
 * Months of the year are identified by the integers 1-12;
 * because this numeric representation for months is in
 * common use, no special constants are defined.
 */

#define Sunday     0
#define Monday     1
#define Tuesday    2
#define Wednesday  3
#define Thursday   4
#define Friday     5
#define Saturday   6
```

图5-6 calendar.c

```
/*
 * Function: PrintCalendar
 * Usage: PrintCalendar(year);
 * ---------------------------
 * This procedure prints a calendar for an entire year.
 */

void PrintCalendar(int year)
{
    int month;

    for (month = 1; month <= 12; month++) {
        PrintCalendarMonth(month, year);
        printf("\n");
    }
}

/*
 * Function: PrintCalendarMonth
 * Usage: PrintCalendarMonth(month, year);
 * ---------------------------------------
 * This procedure prints a calendar for the given month
 * and year.
 */

void PrintCalendarMonth(int month, int year)
{
    int weekday, nDays, day;

    printf("    %s %d\n", MonthName(month), year);
    printf(" Su Mo Tu We Th Fr Sa\n");
    nDays = MonthDays(month, year);
    weekday = FirstDayOfMonth(month, year);
    IndentFirstLine(weekday);
    for (day = 1; day <= nDays; day++) {
        printf(" %2d", day);
        if (weekday == Saturday) printf("\n");
        weekday = (weekday + 1) % 7;
    }
    if (weekday != Sunday) printf("\n");
}

/*
 * Function: IndentFirstLine
 * Usage: IndentFirstLine(weekday);
 * --------------------------------
 * This procedure indents the first line of the calendar
 * by printing enough blank spaces to get to the position
 * on the line corresponding to weekday.
 */

void IndentFirstLine(int weekday)
{
    int i;
```

174

图 5-6（续）

```
string MonthName(int month)
{
    switch (month) {
      case  1: return ("January");
      case  2: return ("February");
      case  3: return ("March");
      case  4: return ("April");
      case  5: return ("May");
      case  6: return ("June");
      case  7: return ("July");
      case  8: return ("August");
      case  9: return ("September");
      case 10: return ("October");
      case 11: return ("November");
      case 12: return ("December");
      default: return ("Illegal month");
    }
}

/*
 * Function: IsLeapYear
 * Usage: if (IsLeapYear(year)) . . .
 * ----------------------------------
 * This function returns TRUE if year is a leap year.
 */

bool IsLeapYear(int year)
{
    return ( ((year % 4 == 0) && (year % 100 != 0))
             || (year % 400 == 0) );
}
```

图 5-6 (续)

小结

在本章中，我们介绍了函数，它使你能用一个简单的名字表示一组完整的操作。利用函数时，程序员可以忽略内部细节，而将注意力集中于函数的整体作用，所以函数提供了减小程序复杂性的关键的工具。

176

本章介绍了最基本的程序设计技术——*逐步精化*的方法。从程序的概要描述开始，通过把概要中的每一片段再划分成更简单的步骤来不断精化每一片段，这样可以通过编写一组函数（其中每一个都相当简单）来解决一个复杂问题。

本章介绍的重点内容包括：

- 函数是由一组程序语句组成，这些语句被放在一起，并被赋予一个名字。程序的其他部分可以调用这些函数，以实际参数的形式传给它一些信息，得到一个由函数返回的结果。
- 每个函数在使用前必须被声明。函数的声明被称为*原型*。
- 除了原型之外，函数还有一个实现，它指出了函数包含的每一个步骤。
- 返回一个值的函数必须有一个`return`语句，它指出了函数的结果。函数可以返回任何类型的值。

的值。常量ϕ被称为黄金比例。古代数学家认为这个比例是从美学上看矩形两条边的最佳比例，但它也经常出现在计算数学中。 178

2. 在中学代数中，我们已经学过标准的二次方程式

$$ax^2+bx+c=0$$

可以用公式

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

求得两个根。第一个根是用“+”替换公式中的“±”，第二个根是用“-”替换公式中的“±”。

写一个C语言程序输入a、b和c三个值，然后计算两个根。如果平方根里面的值是负数，则该方程没有实数解，程序要显示这个信息。可以假设a的值为非0。程序的运行示例如下：

```
Enter coefficients for the quadratic equation:
a: 1↵
b: -5↵
c: 6↵
The first solution is 3
The second solution is 2
```

3. 修改图5-1所示的程序c2ftable.c，使它能将华氏温度转换为摄氏温度。你的程序应该被分为主程序和转换函数，就如c2ftable.c程序一样。在主程序中，华氏温度的范围为32～100，步长为2度，以此产生一个等价的摄氏温度表，小数点后保留一位。输出表的基本格式如下所示：

```
Fahrenheit to Celsius table.
  F     C
 32    0.0
 34    1.1
    ...
 98   36.7
100   37.8
```

4. 第4章的练习8中引入了Fibonacci数列，其中每一项是前两项的和。按该练习的条件重写程序，但要改变实现方式，使主程序调用一个函数Fib(n)计算Fibonacci数列的第n项。若所需计算的次数相同，比较新的实现方法和第4章所用方法哪个效率更高？ 179

5. 写一个函数RaiseInToPower，它取两个整数，n和k，返回n^k。用你设计的函数显示一张2^k的表，k从0～10。

6. 写一个函数RaiseRealToPower，它取一个浮点数x和一个整数k，返回x^k。实现你的函数使它在k为负数时也能正确计算结果，用关系式

$$x^{-k}=\frac{1}{x^k}$$

用你的函数显示所有k为－4～4之间的10^k的表，如下表所示：

使得循环正确地显示值1.0、1.1、1.2等，包括2.0。如4.7.3节指出，由于浮点数精度的限制，在某些机器上这个循环将不会包括值2.0。

11. 在5.3.6节中，下面的循环用来给用户再玩一次游戏的机会（游戏本身没有指定）：

```
while (TRUE) {
    PlayOneGame();
    printf("Would you like to play again? ");
    answer = GetLine();
    if (StringEqual(answer, "no")) break;
}
```

181

该方法的主要问题是仅当用户输入的是no时，循环才会退出。程序将其他的响应（如NO或Absolutely not）解释为TRUE，并重新开始游戏。从程序设计的观点来讲，将用户的响应与yes和no比较会更好一些。如果用户给出了两个答案中的一个，程序按答案执行。如果用户给出另外的响应，程序会重新问这个问题。

写一个谓词函数GetYesOrNo，它接收一个提示符串作为参数，将这个提示符显示给用户作为问题。如果用户用单词yes响应，GetYesOrNo将返回TRUE。如果用户用no响应，GetYesOrNo将返回FALSE。如果用户给出任何其他响应，函数会显示一个消息，告诉用户合法的响应值，然后重新提该问题。

给出GetYesOrNo函数，可以将玩游戏的循环改变为：

```
while (TRUE) {
    PlayOneGame();
    if (!GetYesOrNo("Would you like to play again? ")) break;
}
```

它会产生下面的运行示例：

```
. . . play the game . . .
Would you like to play again? yes↵
. . . play the game . . .
Would you like to play again? maybe↵
Please answer yes or no.
Would you like to play again? yes↵
. . . play the game . . .
Would you like to play again? no↵
```

12. 本书中Combinations函数的值通常以下图所示的三角形形式输出

C(0,0)

C(1,0) C(1,1)

C(2,0) C(2,1) C(2,2)

C(3,0) C(3,1) C(3,2) C(3,3)

C(4,0) C(4,1) C(4,2) C(4,3) C(4,4)

这个图称为Pascal三角形，以发明它的17世纪法国数学家Blaise Pascal命名。Pascal三角形有一个有趣的特性，它的每一个内部项是它上面两个项之和。

182

写一个C语言的程序，显示如下所示的Pascal三角形的前8行：

第6章 算 法

People like definite decisions,
Tidy answers, all the little ravellings
Snipped off, the lint removed. . . .

— Gwendolyn Brooks, *Annie Allen*, 1949

学习目标

- 通过研究经典数学中的一些算法，深入理解算法形成过程。
- 了解大多数问题都是可以使用不同的算法来解决。
- 掌握在不同算法中选取某个算法时需要考虑的问题。
- 掌握如何确定算法是正确的。
- 理解并能应用brute-force算法。
- 能够用连续逼近的技术编写程序。
- 学会用`Error`函数向用户报告错误情况。
- 能够编写一个完成级数展开的程序。

184
~
185

函数对于程序设计来说是很重要的，其中的一个原因是函数提供了算法实现的基础。算法本身是抽象的策略，通常用自然语言表达。函数是以某种程序设计语言表示的算法的具体实现。当你要将算法作为程序的一部分实现时，通常要写一个函数来执行该算法，而该函数也可以调用其他函数处理它的一部分工作。

在实例程序中，你已经见过一些简单的算法，但还没有深入研究算法过程本身的特性。到目前为止，你所见过的大多数程序设计问题都太简单，马上就可以找到解决方法。然而，随着问题越来越复杂，需要周密地思考才能找到解决的策略，所以在写最终的程序前，需要考虑多个策略而不是一个策略。

本章将从解决几个经典数学中的问题开始，说明算法策略的具体形成过程，每个问题都可以用几种方法解决。通过用多种解决方案解决一个问题，可以学会如何比较不同的策略，以及如何选取最佳的策略。

6.1 测试素数

在西方，有记录可查的第一个数学家是约4000年前埃及的教堂中的一个牧师。从那时起，数学不断地发展，在古希腊进入了一个黄金时代，产生了许多数学家，如欧几里德、毕达哥拉斯、泰利斯、阿基米德以及希帕蒂娅。这些希腊数学家在许多不同的数学领域各有所长，但似乎都对研究非负整数的特性特别感兴趣，这就是现在被称为**数论**（number theory）的数学领域。

对希腊数学家和今天的数学家来说，数论中的一个重要问题是确定一个数是否为素数。如果一个正整数n只能被1和它本身整除，则这个正整数n就是**素数**（prime）。例如，23是一个素数，因为除了1和23以外，没有数能整除它。而35就不是素数，因为除了1和35之外，它还能被7和5整除。注意，按照这个定义，整数1不是素数，因为它只被1整除，这一点很重要。

第三，IsPrime会在有限步后结束吗？通过检查程序的结构就可以回答此问题。函数中仅有的长期运行的部分是一个for循环。从控制行可以看出，每次调用IsPrime函数，循环将执行n个周期。如果n非常大，函数会执行很长时间，但最终是会返回的。

因为满足这三个条件，所以图6-1中给出的函数IsPrime确实是一个算法。

6.1.3 说明IsPrime算法的正确性

除了确定该实现是一个算法外，确定算法是否正确也是很重要的。对每一个可能的n值，它都能给出正确的答案吗？很难用任何形式化的方法证明算法的正确性，进行这种证明所用的大多数技术超出了本书的范围。即使这样，对IsPrime的实现还是可以概述其证明的方法。

回想一下程序中的变量divisors的意义。在for循环的每个周期中，divisors记录了到这点为止遇到的约数个数。在for循环开始前，还没有发现任何约数，因此divisors被初始化为0。在循环的每一个周期中，当发现一个新的约数，程序将divisors加1。因此，在第i个周期结束时，divisors保存了1到i之间的约数个数。因为对每个循环值i，这个特性都为真，所以在第n个周期结束时也为真，于是变量divisors包含了1到n之间的约数个数。如果约数个数是2，则n一定是素数。

在循环开始时为真，在每一个循环周期结束前也为真的特性称为**循环不变式**(loop invariant)。对简单程序来说，通过说明维护了相应的循环不变式，可以容易地说明程序的正确性。对复杂的程序来说，证明正确性是相当困难的。在这些情况下，很少能依赖一些形式化的方法来确定程序是否确实递交了正确的答案。 188

通常，有两种方法可以增加你对程序正确性的把握。第一，按着程序代码一步一步执行，确信程序的行为和你的要求一致。这个过程被称为**桌面检查**（desk-checking）。学会用怀疑的眼光执行桌面检查并找到推理程序的技术需要实践和训练。第二种方法称为**测试**（testing），即利用尽可能多测试实例运行你的程序，对每种情况都检查执行结果是否正确。例如，可以写一个主程序，对1～1000之间的数调用IsPrime，然后检查输出，确信所有的数据都被正确分类，这样你对算法的正确性就会更有把握。

然而，完全严格的测试几乎是不可能的，记住这一点很重要。要测试的情况是不计其数的。当代著名的计算机科学家Edsger Dijkstra曾说过："测试能发现错误的存在，但永远不能发现它们不存在"。当你编写一个较大的复杂的程序时，必须把这个原理牢记在心。

常见错误 当你写一个程序时，尽量彻底地测试程序是很重要的。尽管这样，程序中还会有许多测试没有发现的错误，因为通常不可能将每一种可能出现的情况都测试一下。

6.1.4 改进算法的效率

和许多程序设计的问题一样，确定一个数是否为素数可以用许多种方法。图6-1所用的策略无疑有很多缺点。特别是该算法的效率比较低，如果你要检查的数字很大，它就不太可行了。例如，如果你对数字1 000 000调用IsPrime，该函数要检查1～1 000 000之间的一百万个数，确定它们是否为约数。测试所有的数是愚蠢的，因为1 000 000明显不是素数；显然，它能被2整除。可以确定的是，一定存在更好的方法，不用检查所有的数就能得到答案。

有多种修改这个基本的算法的方法，可以改善IsPrime函数实现的效率。例如，下面的三种修改方案都能明显改善IsPrime执行的效率：

1) 从1 000 000这个例子可知，IsPrime没有必要检查所有的约数。只要发现任何大于1

是无效的。

这种问题是很难发现的，它存在于for循环的控制行中：

```
for (i = 3; i<=sqrt(n); i+=2)
```

正如本章后面将介绍的，现代计算机能在相当短的时间内计算平方根。尽管如此，计算平方根还是比完成简单的算术运算（如乘和除）花的时间更长。如程序所示，IsPrime在每个for循环周期中都要调用sqrt函数，尽管每一次循环返回的结果都是相同的。因为n的值在循环内部没变过，sqrt(n)的值也不会变。为了避免一次又一次地调用sqrt，而得到的是同一个结果，可以在循环开始前先计算出sqrt(n)，并把它存入一个变量。例如，可以引入一个double类型的变量limit，用下列两个语句替换for循环的控制行：

```
limit = sqrt(n);
for (i = 3; i<= limit; i+=2)
```

这个简单的改变明显地改善了IsPrime实现的效率。

常见问题 *当在某一程序中使用循环时，检查一下是否在循环中存在某些放在循环开始前执行更好的计算。如果存在，则可以计算一次结果，把它存在一个变量里，然后在循环中使用这个变量，这样可以改善程序的效率。*

这个实现还有一个更难看出的问题。发现这个逻辑错误是很难的，因为它可能在测试中不出现。你可能用成千上万个输入值测试这个函数，每次得到的都是正确的答案。但某些人仍然可以用以前没有测试过的例子使函数失败。更令人烦恼的是，IsPrime的这个实现在一台机器上可能给出正确答案，而在另一台机器上却给出不正确的答案。

为了理解这个问题，可以回顾一下4.7.3节中的讨论。在这一节中，我们说过对浮点数判断严格的相等是很危险的。假设数n是某一个素数的平方。例如，假设n是49，它是7的平方。当对49调用sqrt函数时，它返回的值是什么？在严格的数学领域中，这个平方根是7，但计算机不是在这个领域中运作。浮点数仅仅是近似，sqrt(49)返回的可能是6.999999999999。尽管 191
这个数比7小一点点，但这个差别足以影响测试i<=limit的结果。如程序中所示，这个测试需要数学上完全精确，用浮点数不能保证这一点。如果i是7，而limit是6.999999999999，则循环的最后一个周期将不会被执行，程序永远不会检查n是否可以整除7。因为7是49唯一的因子，不检查7是否为49的因子意味着49将被错误地分类为素数。另一方面，如果sqrt(49)返回的是7.0或7.000000000001，IsPrime将会给出正确的答案。因此，这个实现的正确性取决于硬件如何执行浮点数运算。

让实现的正确性依赖于运行它的计算机的特性是一个严重的错误。这个问题很容易解决，可以修改这个程序使之与机器的精度无关。如果n的平方根小于某一界限，为了确保起见，函数总是多检查一个可能的约数。多测试一个约数没什么害处，这是确保答案正确性所付出的很小的代价。因此，要改正程序，所需要做的只是修改limit的赋值语

```
bool IsPrime(int n)
{
    int i, limit;

    if (n <= 1) return (FALSE);
    if (n == 2) return (TRUE);
    if (n % 2 == 0) return (FALSE);
    limit = sqrt(n) + 1;
    for (i = 3; i <= limit; i += 2) {
        if (n % i == 0) return (FALSE);
    }
    return (TRUE);
}
```

图6-3 IsPrime的正确实现

用这个策略计算GCD的函数如图6-4所示。

再重申一次，你必须问你自己几个有关实现的问题。GCD的brute-force的实现是不是总能给出正确的答案？它是不是总能结束，函数是否可能永远运行？

```
int GCD(int x, int y)
{
    int g;

    g = x;
    while (x % g != 0 || y % g != 0) {
        g--;
    }
    return (g);
}
```

图6-4 GCD的brute-force的实现

要知道程序是否总能给出正确的答案，需要看一下while循环中的条件

```
x % g != 0 || y % g != 0
```

while的条件指出在什么情况下循环将继续。为了找出什么样的条件会使循环终止，必须把while的条件取反。对一个包含&&或||的条件取反可以用德·摩根定理来完成，这个定理在第4章的4.3.2节已经介绍。德·摩根定理指出在while循环出口处，下列条件必须成立：

```
x % g == 0 && y % g == 0
```

从这个条件可以立即看出，g的最终值一定是公约数。要确定这事实上就是最大公约数，必须考虑嵌在while循环体中的策略。策略中的关键因素是该程序反向测试所有的可能性。GCD 194
永远不可能大于x（或y，对于这个问题），brute-force从这个值开始搜索。如果程序一旦出了while循环，它肯定已经测试了x和g的当前值之间的每一个值。因此，如果有一个较大的值能被x和y整除，那么程序在较早的while循环的迭代中就会发现这个数。

为了说明函数会终止，必须首先认识到即使没有找到最大公约数，g的值最终会到达1。此时，while循环肯定会终止，因为不管x和y有什么值，1总能被x和y整除。

6.2.2 欧几里德算法

然而，brute-force不是一个有效的策略。事实上，如果只关心效率的话，上一节中提到的GCD实现的brute-force算法是一个很差的选择。例如，考虑一下对1 000 005和1 000 000调用这个函数，会发生什么。brute-force在找到5之前将运行100万次while循环的循环体，其实这个答案只要考虑两个数就能很容易地确定了。

你所需要考虑的是找出一个算法，它比brute-force所需的执行步骤少，但却保证能结束，并返回正确的答案。这是聪明性和容易理解所付的代价。幸运的是，所需的具有创造性的想法已由古希腊的数学家欧几里得提出来了，他的*Elements*（book 7, proposition II）这本书中包含了一个解决这个问题的非常出色的算法。用自然语言，我们可以把欧几里德算法描述如下：

1) 取x除以y的余数，称余数为r。

2) 如果r是0，过程完成，答案是y。

3) 如果r非0，设x等于原来y的值，y等于r，重复整个过程。

你可以将这个算法翻译成GCD函数的实现，如图6-5所示。

```
int GCD(int x, int y)
{
    int r;

    while (TRUE) {
        r = x % y;
        if (r == 0) break;
        x = y;
        y = r;
    }
    return (y);
}
```

图6-5 GCD的欧几里德算法

GCD函数的这个实现同样能正确地找出两个数的最大公约数。它与brute-force的实现有两处不同。一方面，它能更快地计算出结果。另一方面，很难证明它的正确性。

在本章介绍的两个实例（检测素数和计算最大公约数）中，算法的选择对解决方案的效率有着极大的影响。在第17章中，我们还可以学到如何量化算法的效率。

6.3 数值算法

在现代世界中，计算机被应用到许多领域。结果，计算机程序员要编写许多不同类型的程序，程序中涉及各种不同的计算操作。有些程序员可能不用任何如平方根这样的数学操作，因此他可以根本不知道math库的内容。甚至对于那些使用平方根的程序员来说，大多数程序员也只满足于使用sqrt函数，而不考虑它的内部实现。然而，他们可以这样做的原因是某些程序员事先已经花时间和精力写了sqrt函数。计算机用来实现如sqrt这样的数学函数的技术称为**数值算法**（numerical algorithm），它是计算机科学的一个重要领域。 197

因为平方根在几何中频繁出现，所以数学家在2000多年的时间里依然对如何计算平方根的问题十分感兴趣，他们已经开发了解决这个问题的许多种算法。为了说明这些方案的技术（同时考虑数值算法的更一般的特性），在6.3.1节和6.3.2节中给出了计算平方根函数问题的两种不同的方法。

6.3.1 连续逼近

解决数值问题的最通用的策略之一是连续逼近的技术。**连续逼近**（successive approximation）是找出近似答案的一般策略，它由下列步骤组成：

1) 先对答案进行猜测。

2) 一旦有了猜测值，就可以用猜测值产生一个更佳的值。例如，假设你测试了你的答案，发现太大。你可以把它变得小一点，用这个小的值作为新的猜测值。相反，如果你原先猜测的值太小，那么可以选择一个较大的值作为下一个猜测值。

3) 如果你能保证，你的每一轮猜测与真实答案越来越接近，那么重复这个过程，最终将会产生一个足以满足应用需要的猜测值。

连续逼近技术的困难部分是第二步中的选择一个新的猜测值，使它满足第三步开始的条件。选择新的猜测值的策略必须考虑特定问题的一些知识。你所需要找到的是一系列的猜测值，每一个猜测值都比前一个猜测值更接近答案，随着序列的继续产生，可以使猜测值与答案达到任意接近程度。如果随着计算的项越来越多，一系列的值会接近于一个极限，那么这个数列被称为**收敛**（converge）的。

用连续逼近方法解决平方根问题的策略是由艾萨克·牛顿在17世纪发明的。计算平方根的牛顿法最好用一个实例来说明。假设要找出16的平方根。你可以随便做一个猜测。但必须小于16。你可以从8开始。你的答案是太大还是太小？如果8是平方根，那么8乘8应该是16。但8乘8是64，对16的平方根来讲，8太大了。你需要选择一个较小的数。但到底小多少？有没有使用这个问题的某些性质来选择下一个猜测值的方法？ 198

牛顿的观点是正确的平方根一定介于当前的猜测值和原先的值除以猜测值的结果之间。在这个实例中，猜测值是8，16除以8是2。也就是说，8太大了，2太小了。这两个值被认为是位于正确答案的两侧。为了保证更精确的猜测，可以取两个值的平均值。结果将更接近于16的平方根，虽然仍然是一个近似值。在这个实例中，将8和2的平均值5作为新的猜测值。

此时，只要简单地重复这个过程即可。16除以5是3.2。平均5和3.2产生一个新的猜测值4.1，它更接近于正确答案。如果你继续这个过程，后两个猜测值是4.001219512和4.00000018584。通过第四个猜测值，牛顿法已经产生了一个非常接近于正确答案的值。这个过程永远也不

6.3.2 报告错误

程序有很多种方法报告执行过程中出现的错误情况。本书中的程序用Error函数向用户报告错误，该函数在genlib库中定义。像printf一样，Error函数取一个控制字符串后跟另一个参数。当Error被调用时，显示一个字符串“Error:”，后跟一个控制字符串，像printf一样，格式代码的地方用一个值替换。在行尾，Error自动包括一个换行符，将光标移到下一行的开始。

Error和printf的主要区别在于Error不用返回到它的调用程序。一旦显示错误消息，整个程序就终止，就像它执行完主程序中的所有语句一样。因此，如果用-1调用Sqrt，屏幕上将会出现下面的输出：

199 ~ 200

```
Error: Sqrt called with negative argument -1
```

一旦出现错误消息，程序停止运行，不再执行任何语句。

在程序中响应一个错误的过程称为**错误处理**（error handling）。使一个程序输出一个错误消息并停止执行并不是特别复杂的错误处理策略。在如sqrt之类的库函数中，最好能提供一些调用程序可以用来采取正确行为的机制。但是，有关最好的错误处理策略的内容超出了本书的范围。当错误情况出现时，程序作出某些反应是非常重要的。如果不检查这些情况，程序可能给出不正确的答案，或根本不能产生答案。例如，如果Sqrt不检查负数参数，那么为了估计一个不存在的答案，程序将进入一个无限循环。使一个程序报告错误并终止是比使它永远运行更有用的响应。

6.4 级数展开

虽然牛顿法是计算平方根问题的一种可行的方法，但还有许多同样有效的方法。其中某些方法可以容易地应用在其他问题中。应用最广泛的方法之一是**级数展开**（series expansion），在这种方法中，函数的值通过对级数项的求和来估计。如果每加入一个新的项都会使总和更接近于一个所期望的值，则级数是收敛的，可以用此级数估计结果。

6.4.1 Zeno悖论

为了说明级数展开的思想，本节再一次探索希腊数学家发现的问题。在公元前5世纪，埃里亚的哲学家Zeno发明了悖论，他建议该行为是不可行的。假设你想穿过一个房间。要做到这一点，首先必须走到一半的地方。从这个地方开始，再走过余下的一半，这意味着再走过原始距离的四分之一。从这里开始，必须再走总距离的八分之一到下一个中点，一直这样走下去。因为余下的距离总是一个能被一分为二的量，所以Zeno认为，这个过程将永远不会结束，你永远不可能到达目的地。这个论点称为**Zeno悖论**（Zeno's paradox）。

201

在Zeno悖论中，横穿房间这个过程中的每一步都是走到剩余距离的一半。如果把Zeno问题以数学形式表示，那么已经走过的每段距离的总和可以表示为下式：

$$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}+\cdots$$

Zeno正确地观察到在这个级数中有无限个项[⊖]。但Zeno在公元前5世纪时无法认识到无限级数

⊖ 在一个数学的级数中，被加起来形成总和的每一个值称为项。由于级数中的项本身可以涉及一些数学运算（如Zeno计算中每个项所用的分数标记），所以级数中所用的项的定义与第2章表达式的讨论中所用的项的定义有所不同。

while循环的有效策略。如果用

```
while (sum != sum + term)
```

取代无限循环中的while控制行，那么只要term加到sum上时产生了一些可注意到的影响，那么循环将继续。

计算Zeno级数和的完整的主程序如图6-7所示。

```
main()
{
    double sum, term;

    sum = 0.0;
    term = 0.5;
    while (sum != sum + term) {
        sum += term;
        term /= 2;
    }
    printf("The sum of Zeno's series is %g\n", sum);
}
```

图6-7 Zeno级数和的主程序

6.4.2 用级数展开法设计平方根函数

和许多数学函数一样，平方根函数也有一个相关的幂级数，这样就可以用前一节介绍的求和技术产生近似结果。但是，找出使用哪一个幂级数需要熟悉微积分。一旦得到级数的公式，编写这样一个程序不费吹灰之力。

如果你不熟悉微积分，跳过下一节，从6.4.4节重新开始。如果你掌握了微积分，可以再来读这一节，看一下计算数学家们是如何提出用于实现数学库中的函数所需的技术。

6.4.3 估计平方根的泰勒级数展开（可选）

最有用的幂级数之一是泰勒（Taylor）级数，一般在大学一年级学习微积分时会遇到这个级数。18世纪的数学家布鲁克·泰勒发现，至少对一定范围之内的参数值，连续可导的函数可以用以下公式近似表示：

$$f(x) \cong f(a)+f'(a)(x-a)+f''(a)\frac{(x-a)^2}{2!}+f'''(a)\frac{(x-a)^3}{3!}+\cdots+f^{(n)}(a)\frac{(x-a)^n}{n!}$$

在这个公式中，符号a表示一个常量，记号f'、f''和f'''表示函数f的一阶、二阶和三阶导数，以此类推。这个公式称为**泰勒公式**（Taylor's formula）。

考虑f是平方根函数的情况。在数学中，取x的平方根与求x的1/2次幂是一样的，因此$f(x)$可以定义为

$$f(x)=x^{\frac{1}{2}}$$

为了找出f'，你需要做的就是用x^c求导的通用公式

$$cx^{c-1}$$

其中c为任意常数。在平方根函数中，c是1/2，因此一阶导数是

$$f'(x)=\frac{1}{2}x^{-\frac{1}{2}}$$

然后，运用同样的规则产生另外的一些导数。每一次都用前一次系数乘以前一次的指数，指

每一项与前一项有什么不同？为了回答这个问题，我们认为每一项都有三个独立的部分：

$$\texttt{corff}\frac{\texttt{xpower}}{\texttt{factorial}}$$

例如，对t_3来说，coeff是3/8，xpower是$(x-1)^3$，factorial是3!。然后，这个问题就变为研究这些部分是如何随项的变化而变化的。

例如，考虑xpower部分。将当前的xpower乘以$x-1$就得到了下一项的xpower。把当前的xpower值保存在一个变量中，就能确保在每个循环周期中通过一次乘法更新这个当前值。

factorial部分怎么办呢？对第i项，这个部分的值总是$i!$。阶乘有一个非常有用的特性：任何数i的阶乘是i乘以比它小的那个数的阶乘。这个特性可以很容易地用它的定义进行证明：

$$\begin{aligned}(i+1)!&=(i+1)\times i\times(i-1)\times(i-2)\times\cdots\times 2\times 1\\&=(i+1)\times i!\end{aligned}$$

因此，为了准备第（$i+1$）个循环周期，所需要做的就是把当前的factorial乘以$i+1$。

唯一剩下的部分是coeff。和其他几项一样，它也是由乘积产生的[⊖]。一般来讲，为了得到第（$i+1$）个循环周期的coeff值，只需要简单地把第i个周期的值乘以

$$\frac{1}{2}-i$$

206

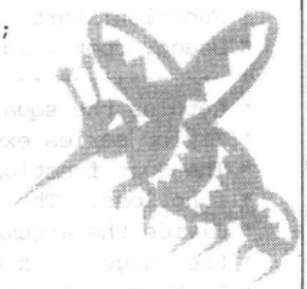
你可以用这些公式从每个部分的当前值中计算出下一项中各部分的值。如果变量coeff、xpower和factorial包含当前项的这些部分，那么为了计算下一项中这些部分的值，所需要做的只是将下列语句作为函数的主循环：

```
coeff *= (0.5 - i);
xpower *= (x - 1);
factorial *= (i + 1);
```

正如计算Zeno级数求和的程序一样，主循环必须执行到加入一个项的值不会改变总和的值为止。写一个循环测试这个条件，并初始化所有的变量，产生如图6-8所示的函数TSqrt。

```
double TSqrt(double x)
{
    double sum, factorial, coeff, term, xpower;
    int i;

    factorial = coeff = xpower = 1;
    sum = 0;
    term = 1;
    for (i = 0; sum != sum + term; i++) {
        sum += term;
        coeff *= (0.5 - i);
        xpower *= (x - 1);
        factorial *= (i + 1);
        term = coeff * xpower / factorial;
    }
    return (sum);
}
```



（如果$x\geqslant 2$，则函数失败）

图6-8 TSqrt的泰勒级数展开

⊖ 详细理解系数的逐项变化需要知道这些项是如何用泰勒公式计算的。如果你读了6.4.3节，就会知道系数是从平方根函数的导数中产生的。然而，即使你没有阅读这一节，你也可以顺利应用这个简单的公式。

```
    if (x == 0) return (0);
    if (x < 0) Error("Sqrt called with negative argument %g", x);
    correction = 1;
    while (x >= 2) {
        x /= 4;
        correction *= 2;
    }
    return (TSqrt(x) * correction);
}

/*
 * Function: TSqrt
 * Usage: root = TSqrt(x);
 * -----------------------
 * Returns the square root of x, calculated by expanding
 * the Taylor series around a = 1, as described in the
 * text.  The function is effective only if x is in the
 * range 0 < x < 2.  Term i in the summation has the form
 *
 *                    xpower
 *       coeff   *  ---------
 *                  factorial
 *
 * where coeff comes from ith derivative of the function,
 * factorial is i!, and xpower is the ith power of (x - a).
 * Each of these components is computed from its previous
 * value.
 */

double TSqrt(double x)
{
    double sum, factorial, coeff, term, xpower;
    int i;

    factorial = coeff = xpower = 1;
    sum = 0;
    term = 1;
    for (i = 0; sum != sum + term; i++) {
        sum += term;
        coeff *= (0.5 - i);
        xpower *= (x - 1);
        factorial *= (i + 1);
        term = coeff * xpower / factorial;
    }
    return (sum);
}
```

图 6-9 (续)

209

6.5 指定数值类型的大小

当你利用本章所示的这些数值应用时，控制数值数据的范围和精度是很重要的。在某种程度上，这些特性取决于硬件，因为计算机内的所有数据都存储在它的内存系统中。对不同类型的计算机，内存划分单元的大小可以不同。从程序员的观点来看，内部结构不同的影响主要体现在数值型的数据在不同的机器上可能受到不同的限制，这使得写一个可移植的程序非常困难。**可移植**（portable）程序是一个能在不同的计算机系统上以完全相同的方式成功运

6.5.2 无符号类型

在C语言中，整型类型int、long和short都可以在前面加上关键词unsigned。加上unsigned表示一种新的数据类型，这种类型仅允许表示非负值。由于无符号变量不需要表示负数，声明一个无符号类型的变量可以保存两倍的整数值。例如，int类型的最大数值是32 767，那么unsigned int类型的最大数值将是65 535。C语言允许将unsigned int类型缩写为unsigned，大多数使用这种类型的程序员都会遵从这个惯例。

Unsigned int类型的值在printf调用中用%u来显示。要显示unsigned long和unsigned short，可以分别用%lu和%hu。 211

在大多数情况下，不需要使用无符号类型，除非你需要对机器中的内部表示保持非常严格的控制。本书中介绍无符号数只是为了保持内容完整，事实上本书中没有使用这种类型。

6.5.3 浮点类型

和整数一样，浮点数也在一定长度的内存单元中表示，因此也有一定的大小限制。然而，对浮点数来讲，内存的限制影响了它的精度（可用的有效位有几位）以及范围。由于精度是硬件的函数，因此浮点数的精度随着机器的不同而不同。例如，在某台机器上，浮点数可以有10位有效位。在这台机器上，值1.0和1.00000000001是一样的，因为计算机不能精确表示10位之后的数。

与整型数一样，C语言提供了三种类型的浮点数。float是精度最小的，但比double类型占用的内存要少。然而，在近十年中，计算机内存越来越便宜，大多数C语言的程序员现在都习惯于将浮点数表示为更精确的double类型，本书也将遵从这个惯例。在C语言中，所有涉及float类型或double类型值的计算都把它当作double类型来处理；数据类型仅影响数据的存储。

然而，对一些精度要求非常高的计算，double类型的精度可能还不够。对于这些需要非常高的精度的应用，ANSI标准也包括了一种称为long double的类型，但这种类型在实际中很少使用。

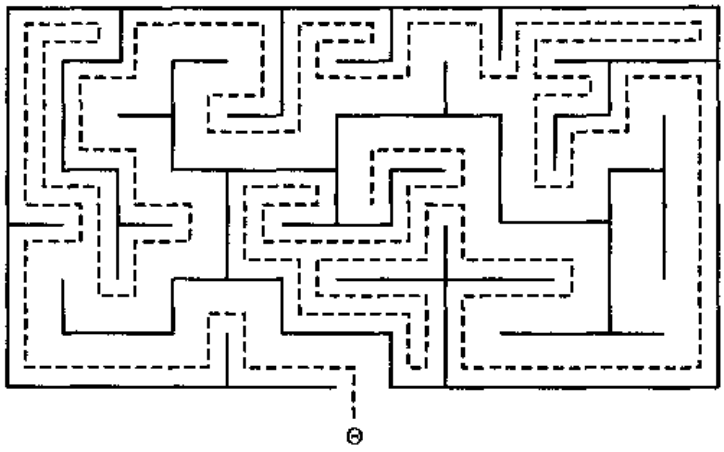
小结

在本章中，介绍了一些设计算法的通用策略。本章出现的实例都选自经典数学，因为数学提供了一些非常有趣的问题，除了数值之外，不需要掌握有关数据类型就能解决这类问题。当你在后面几章学过了其他的一些数据类型后，你会发现这里介绍的算法技术也可以应用于其他类型的问题。

本章介绍的要点包括：

- 解决特定问题通常有几种不同的算法。选择一个最适合应用的算法是程序员任务中最重要的部分。
- 当在不同的算法之间选择时，需要考虑许多不同的因素，包括效率、可维护性和清晰性。选择一个特定的算法需要在这些指标之间进行权衡。 212
- 在改善程序的其他方面时，决不能牺牲正确性。
- 证明程序的正确性是非常困难的任务。在大多数情况下，彻底的测试可以增加你对解决方案正确性的把握。然而，即使通过了良好的测试，还是会有一些隐蔽的错误。
- *brute-force*算法检查了每一种可能性，因而比较麻烦。这样的算法通常容易理解，但效率非常不高。

为了从迷宫中逃出，特修斯将他的右手放在墙上不离开，他走的路径如下图中的虚线所示。



如果你要写一个程序，为一个名叫特修斯的机器人实现右手法则。你可以访问一个包含下列函数的库：

```
void MoveForward(void);   /* 向前移动到下一个正方形区域 */
void TurnRight(void);     /* 右转但不移动 */
void TurnLeft(void);      /* 左转但不移动 */
bool IfFacingWall(void);  /* 如果特修斯面对墙，则返回TRUE */
bool IfOutside(void);     /* 如果特修斯逃离，则返回TRUE */
```

用这些函数写一个过程EscapeFromMaze，实现右手法则所提议的算法。

2. 在许多情况下，知道一个数是否为素数还不够，有时，还需要知道它的约数。每个大于1的正整数都能表示为素数的乘积。这个因数分解是唯一的，被称为**素数分解**（prime factorization）。例如，数字60可以分解为$2 \times 2 \times 3 \times 5$，其中每个约数都是素数。注意，同一个素数在因数分解中可以出现多次。

写一个程序显示数*n*的素数分解。下面是程序运行的一个实例：

```
Enter number to be factored: 60↵
2 * 2 * 3 * 5
```

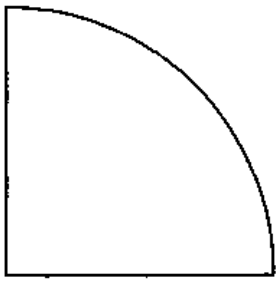
3. 希腊数学家对那些真约数的和等于该数本身的这些数特别感兴趣（*n*的真约数指的是小于*n*本身的任何约数）。这样的数被称为**完全数**（perfect number）。例如，6是一个完全数，因为它是1、2、3的和，1、2、3是小于6但能被6整除的数。类似地，28也是一个完全数，因为它是1、2、4、7、14的和。 215

写一个谓词函数IsPerfect，它取一个整数n，当n是完全数时，返回TRUE，否则返回FALSE。写一个主程序，用IsPerfect检查1～9999中的每一个数，看它们是不是完全数。当发现一个完全数时，将它显示在屏幕上。前两行的输出应该是6和28。此范围中的程序应该还能找到两个完全数。

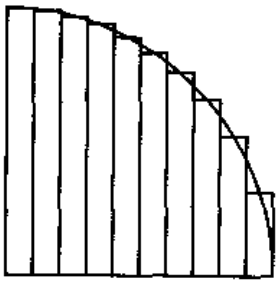
4. 当练习3中得到完全数的程序能够工作后，仔细考虑一下你用的算法。在不牺牲正确性的条件下设计程序的一个新版本，改善它的效率。

5. 修改本书中的牛顿算法，让它计算立方根而不是平方根。将该算法表示为一个函数CubeRoot，它取一个double类型的参数，返回另一个double类型的值，即参数的立方根。

9. 也可以用圆弧围住的面积的近似值计算π的近似值。考虑四分之一个圆：



它的半径r等于2英寸。利用圆面积的公式，很容易就能确定四分之一个圆的面积应该是π平方英寸。这个面积也可以通过将一连串的矩形面积加起来作为近似值，其中，每个矩形都有固定的宽度，其高度满足如下条件：圆弧通过矩形上面一条边的中点。例如，你把这个面积从左到右划分成10个矩形，可以得到下面的图： 217



矩形面积的和给出了一个四分之一圆面积的近似值。矩形越多，近似值越正确。

对每一个矩形来说，宽度w是一个常量，是半径除以矩形个数的结果。另一方面，高度h的值取决于矩形的位置。如果水平方向上矩形的中点位置是x，矩形的高度可用距离公式

$$h = \sqrt{r^2 - x^2}$$

来计算。矩形的面积可以简单地记为$h \times w$。

将它分成100个矩形，写一个程序计算四分之一圆面积。

10. 数学常量e可以用下列幂级数展开来计算：

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \cdots$$

写一个程序通过计算级数的和计算e的值，要求结果达到浮点数的精度限制。显示答案，小数点后取10位有效位。 218

第二部分

库和模块化开发

概要

如果某一天，你在软件行业找到了一份工作，你会发现很少有人完整地写完一个程序。在大多数公司里，程序员都是按团队工作，建立应用，这些应用的规模相当大，不是任何一个人可以单独完成的。使许多程序员能一起工作是现代软件工程最大的挑战。第二部分中的四章将介绍如何把大程序分解成独立的模块，如何设计这些模块，使它们能作为其他应用的库。

接口是库和它的使用者之间交换信息的媒介并给出了交换信息的结构。从概念上讲，程序设计的接口也表示有关边界特性的共识，提供了库的创建者和使用者之间所需的临界信息。 222

例如，考虑第5章引入的math库。math库中定义了一些函数，如sqrt。使用math库的程序员可以调用sqrt计算平方根，而不需要指出计算平方根的每个步骤。这些步骤是平方根函数实现的组成部分，它是由创建math库的程序员编写的。第6章介绍了两种可行的方法实现sqrt函数——牛顿法和泰勒级数展开。库的实现者正是使用这些方法中的某一种，或其他能计算出正确结果的算法。

掌握如何调用和实现sqrt是很重要的。然而，认识到调用函数和实现函数完全不相关也很关键。成功的程序员经常使用那些他们并不知道如何编写的函数。反过来，实现库函数的程序员也不能预见该函数的所有潜在应用。

为了强调实现库的程序员和使用库的程序员之间的区别，计算机科学家采用了特定的术语去谈及处于特定角色的程序员。很自然，实现库的程序员被称为**实现者**（implementor）。因为用户这个词通常指的是运行程序的人，而不是写程序的人，因此调用库函数的程序员被称为这个库的**客户**（client）[⊖]。

尽管客户和实现者对库有不同的看法，但他们都必须了解库设计中的某些问题。作为客户，不需要知道工作的细节，但必须知道如何调用它。而作为实现者，你不用关心客户程序员会如何使用库。但是，你必须给客户提供调用函数所需要的信息。对库中的每一个函数，客户必须知道下列内容：

- 函数名。
- 所需的参数以及参数的类型。
- 返回的结果类型。

这些信息正好是函数原型所提供的信息，这绝不是一个巧合。在C语言中，函数的原型和它的实现是分开的，因为它们传递信息给不同的读者。客户和实现者必须就函数的原型达成一致，这意味着这些信息是接口的一部分。相比之下，只有实现者是关心函数的实现。在接口中包含这个函数的原型，使客户可以使用该函数的过程称为**导出**（exporting）这个函数。 223

客户和实现者之间的关系可以用下图说明：

客户	接口	实现者
负责： • 如何使用函数	双方约定： • 函数原型	负责： • 函数是如何工作的

接口和头文件

在计算机科学中，接口是一个概念性的实体。它是由实现库的程序员和使用库的程序员之间的理解所组成，清楚地说明双方都需要的信息。然而，当你编写一个C语言的程序时，必须有一种方法将概念性的接口表示为实际程序的一部分。在C语言中，接口习惯上被表示为头文件。其实从第2章开始，你已经在使用头文件了，并且已经遇到了几个不同的头文件，包括stdio.h，math.h和genlib.h。每个头文件都指出了一个基本的库的接口。

接口的抽象概念和表示接口的实际的头文件之间的区别似乎很小。在许多方面，这个区别类似于算法和实现此算法的程序。算法是一个抽象的策略；而程序是一个算法的具体实现。

⊖ 在计算机科学中，术语客户有时指使用库的代码，有时指写这段代码的程序员。如果可能会混淆的话，我将用客户代码表示使用库的代码，虽然本书也将遵循标准，但只要意义明确，在这些情况下也可以使用客户这个词。

一对值来标识，通常记为 (x, y)，它指出了该点沿x轴和y轴的位置。这些值被称为点的**坐标**(coordinates)。坐标用相对于原点的英寸计量，原点的坐标为 (0，0)。从原点开始，x的值随着往右移而增加，y的值随着往上移而增加。

图形库中的坐标可以以两种形式出现：

- **绝对坐标** 用相对于原点的坐标值指定点在窗口中的位置。例如，图7-1中的实心点的绝对坐标是 (2.0, 1.5)。
- **相对坐标** 用该点与最后一次指定的位置在每根轴上的距离来指定某一位置。例如，图7-1上的空心点的绝对坐标为 (2.5, 1.5)。然而，如果用它相对于实心点的坐标来表示，这个点移动的相对坐标为 (0.5, 0.0)。如果要将这些点连成一条线，标准的方法是先用绝对坐标指定第一个点，然后用相对的方式指定连接线。

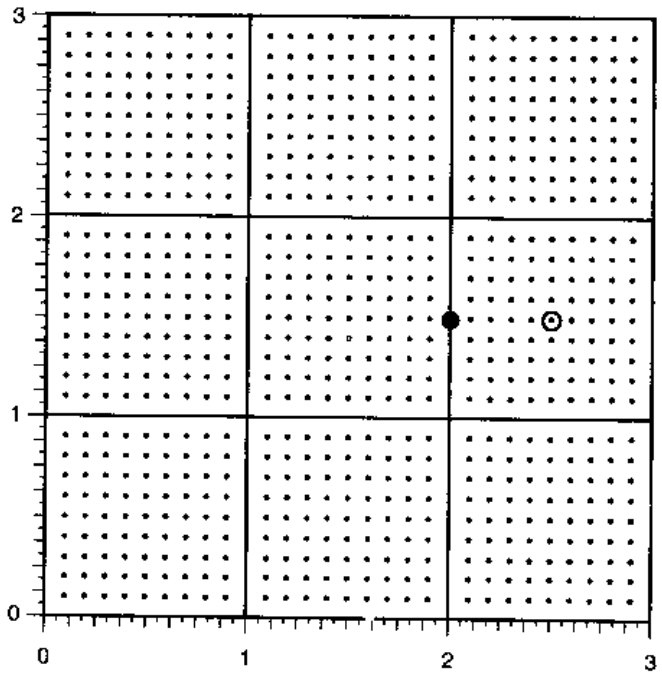


图7-1 图形库中的坐标

在绘图过程中，可以想象成在覆盖在屏幕上的透明图画纸上有一支笔。可以用绝对坐标把笔移到屏幕上的任何位置。一旦到了该位置，可以从当前位置开始，将笔连续在图形纸上移动 226
一个相对位置，画出一条直线。从这个位置开始，又可以画一条从上一个终点开始的直线[⊖]。

7.2.2 graphics.h接口的函数

graphics.h接口仅导出了很少的画图函数和过程。要画出复杂的图片，肯定需要更强大的功能，因为这个接口比较简单，所以我们能在一章中介绍一个完整的图形软件包。你也能很容易地理解它。该图形库包括下列函数：

⊖ 我的一个艺术家朋友告诉我，他们把下面的图案



看成是一条直线，因为它没有断。在本书中，我们用术语线表示一段直线。因此，这个图被认为是由三段头尾相连的线组成。

```
 *   DrawLine(dx, dy);
 *   DrawArc(r, start, sweep);
 *   width = GetWindowWidth();
 *   height = GetWindowHeight();
 *   x = GetCurrentX();
 *   y = GetCurrentY();
 */

/*
 * Function: InitGraphics
 * Usage: InitGraphics();
 * ----------------------
 * This procedure creates the graphics window on the screen.
 * The call to InitGraphics must precede any calls to other
 * functions in this package and must also precede any printf
 * output.  In most cases, the InitGraphics call is the first
 * statement in the function main.
 */

void InitGraphics(void);

/*
 * Function: MovePen
 * Usage: MovePen(x, y);
 * ---------------------
 * This procedure moves the current point to the position
 * (x, y), without drawing a line.  The model is that of
 * the pen being lifted off the graphics window surface and
 * then moved to its new position.
 */

void MovePen(double x, double y);

/*
 * Function: DrawLine
 * Usage: DrawLine(dx, dy);
 * ------------------------
 * This procedure draws a line extending from the current
 * point by moving the pen dx inches in the x direction
 * and dy inches in the y direction.  The final position
 * becomes the new current point.
 */

void DrawLine(double dx, double dy);

/*
 * Function: DrawArc
 * Usage: DrawArc(r, start, sweep);
 * --------------------------------
 * This procedure draws a circular arc, which always begins
 * at the current point.  The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle.  This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
```

图 7-2 (续)

```
#define_graphics_h
```

接口中的最后一行是

```
#endif
```

这些行的作用将在第8章中讨论，第8章主要介绍如何编写一个接口。现在你可以忽略这些行。虽然它们对编译器来说是很重要的，但对理解接口是如何工作的却毫无用处。

227 ~ 230

接口的其余部分仅由注释和函数原型组成。在剩余部分中，注释占头文件的90%。即使编译器忽略这些注释，但它们仍是接口中最重要的部分。接口实际的读者不是编译器，而是准备编写客户代码的程序员。注释的作用是帮助程序员全面理解这个抽象并使用接口提供的功能。

7.2.3 软件包初始化

graphics.h接口中的第一个过程是InitGraphics。正如接口中的注释所指出的，这个过程初始化图形库，必须在软件包中的其他函数和用printf在屏幕上显示任何输出以前调用它，库软件包需要一些初始化是很普遍的。当使用一个接口时，先把它通读一遍，看看是否需要初始化是一个比较好的策略。

7.2.4 画直线

函数MovePen和DrawLine是图形库提供的主要的画线工具。首先，我们从点 (0.5, 0.5) 向上画一条一英寸的直线。在任何使用图形库的主程序中，第一步总是

```
InitGraphics();
```

要画一条线，要先把笔移到点 (0.5, 0.5)：

```
MovePen (0.5, 0.5);
```

从这里开始，所有要做的就是画一条在*x*轴上不变，在*y*轴上向上移一英寸的线：

```
DrawLine (0.0, 1.0);
```

完整的程序如图7-3所示。

```
/*
 * File: oneline.c
 * ---------------
 * This program draws a single straight line.
 */

#include <stdio.h>
#include "genlib.h"
#include "graphics.h"

main()
{
    InitGraphics();
    MovePen(0.5, 0.5);
    DrawLine(0.0, 1.0);
}
```

图7-3 oneline.c

常见问题 确保任何使用图形库的程序的第一行要调用InitGraphics函数。作为一条规则，应该牢记库通常需要初始化。因此，应该检查每一个接口，看看是否需要进行初始化。

```
/*
 * Function: DrawArc
 * Usage: DrawArc(r, start, sweep);
 * --------------------------------
 * This procedure draws a circular arc, which always begins
 * at the current point.  The arc itself has radius r, and
 * starts at the angle specified by the parameter start,
 * relative to the center of the circle.  This angle is
 * measured in degrees counterclockwise from the 3 o'clock
 * position along the x-axis, as in traditional mathematics.
 * For example, if start is 0, the arc begins at the 3 o'clock
 * position; if start is 90, the arc begins at the 12 o'clock
 * position; and so on.  The fraction of the circle drawn is
 * specified by the parameter sweep, which is also measured in
 * degrees.  If sweep is 360, DrawArc draws a complete circle;
 * if sweep is 90, it draws a quarter of a circle.  If the value
 * of sweep is positive, the arc is drawn counterclockwise from
 * the current point.  If sweep is negative, the arc is drawn
 * clockwise from the current point.  The current point at the
 * end of the DrawArc operation is the final position of the pen
 * along the arc.
 *
 * Examples:
 *   DrawArc(r, 0, 360)     Draws a circle to the left of the
 *                          current point.
 *   DrawArc(r, 90, 180)    Draws the left half of a semicircle
 *                          starting from the 12 o'clock position.
 *   DrawArc(r, 0, 90)      Draws a quarter circle from the 3
 *                          o'clock to the 12 o'clock position.
 *   DrawArc(r, 0, -90)     Draws a quarter circle from the 3
 *                          o'clock to the 6 o'clock position.
 *   DrawArc(r, -90, -90)   Draws a quarter circle from the 6
 *                          o'clock to the 9 o'clock position.
 */

void DrawArc(double r, double start, double sweep);
```

图7-4 DrawArc的接口项 233

例如，此文档建议可以调用

```
DrawArc (r, 0, 360);
```

在当前点的左边画一个完整的圆。

这条弧半径为r，从0度开始，即3点钟的位置。它延伸360度，因此创建了一个完整的圆。弧的开始位置是调用时笔的当前位置。相对于所画的圆的位置，这个位置在最右边，这个圆位于左边。

基于上述讨论，应该很容易编写一个画半径为半英寸，圆心在点（1, 1）的圆。所需要做的只是将笔移到圆最右边的起始点，然后调用DrawArc。主程序为

```
main()
{
   InitGraphics();
   MovePen(1.5, 1.0);
   DrawArc(0.5, 0, 360);
}
```

graphics.h接口的一部分并不成问题。因为C语言提供了自定义函数的功能，你可以自己创建这些工具。

7.3.1 定义DrawBox

为了说明这个过程，假设要定义一个过程DrawBox，它可以沿着坐标轴的方向画一个矩形框。写DrawBox的第一步是定义它的原型。这是设计的思考过程。你知道了过程的名字，但还必须考虑包含的参数。确定所需要的参数的最有用的策略是确定这个实现需要什么信息。你不能只给出一个等价于自然语言的命令："画一个矩形"。实现应该知道矩形有多大，把它放在屏幕的什么地方。获取这些信息的常用方法是由客户用参数的形式提供。

尽管如此，设计DrawBox过程还是有多种方法。一种可能的设计方法是仅用两个参数（width和height）指出方框的大小。为了指出方框的位置，应该调用MovePen。调用了MovePen后，就可以以此位置为参照来画方框。因此，在位置（x，y）画一个框，你应该写

```
MovePen(x, y);
DrawBox(width, height);
```

（这个实例不是最终的设计）

另一种DrawBox的设计是采用四个参数，x、y、width和height，它将设定位置和设定大小组合在一起，如下所示：

```
DrawBox (x, y, width, height);
```

因为第二种形式对调用程序来说更方便，通常都采用第二种形式，但两种设计方法都是可行的。

除了确定所需的参数个数外，还必须指明前两个参数的解释。在（x，y）位置画一个方框是什么意思？方框还没有一个明确的起始点。相对于矩形框，点（x，y）在哪里？对某些应用来说，比较方便的方法是使点（x，y）表示方框的中心。然而，更常用的策略是将方框的起始点定义在它的左下角，就如图形窗口的左下角是整个坐标系的原点一样。于是，点（x，y）指出了起始的位置。不管如何定义点（x，y）和方框的关系，需要做的主要工作是确保函数的文档使你的设计决策显得很清晰。

因此，DrawBox的一种可能的原型是

```
void DrawBox (double x, double y, double width, double height);
```

其中，x和y指出了方框的起始位置，width和height指出了它的大小。由于这个过程是自己创建的，而不是库的一部分，所以还需要定义它的实现。这个实现由按参数值画四条线所必需的步骤组成。在本章中，你已经画过一个方框，现在所需要做的只是将直接的坐标值转换为更一般的基于参数的形式：

236

```
void DrawBox(double x, double y, double width, double height)
{
    MovePen(x, y);
    DrawLine(0, height);
    DrawLine(width, 0);
    DrawLine(0, -height);
    DrawLine(-width, 0);
}
```

该实现先移到方框的原点，然后画形成该方框所需的四条线段。

现在你已经编写出这个过程，就可以改变在屏幕上画方框的程序的实现，使它用你的新工具。结果程序drawbox.c如图7-5所示。

7.3.2 定义DrawCenteredCircle

定义一个画完整的圆的函数是非常有用的。因为圆没有角，定义一个画相对于圆心的圆的函数DrawCenteredCircle是很有意义的。这个函数需要三个参数：圆心的x和y轴位置以及它的半径r。因此，DrawCenteredCircle的原型为

```
void DrawCenteredCircle (double x, double y, double r);
```

它的实现为

```
void DrawCenteredCircle(double x, double y, double r)
{
    MovePen(x + r, y);
    DrawArc(r, 0, 360);
}
```

虽然你也可以直接用DrawArc，但似乎DrawCenteredCircle更是裁剪得适合用户需求。首先，完整的圆在图形中相当普通，比一段弧出现的机会要多得多。另一方面，用高级函数可以使你从记住DrawArc如何解释角度的过程中解脱出来，而这些问题在画一个圆的时候是不需要考虑的。DrawCenteredCircle提供了方便性和简单性，这两点在程序设计中都有实用价值。

237 ~ 238

7.3.3 绝对坐标和相对坐标间的切换

MovePen过程用绝对坐标指出线的起点，然后用DrawLine通过相对坐标画线。对某些应用，将笔移到相对于原始位置的新位置，但不要画线也是很有用的。反过来，有时画一条到特定的绝对坐标位置的线也是很有用的。

函数GetCurrentX和GetCurrentY提供了写一个MovePen的相对版本和DrawLine的绝对版本的方便性。新的函数称为AdjustPen和DrawLineTo，它们的实现如图7-6所示。

```
/*
 * Function: AdjustPen
 * Usage: AdjustPen(dx, dy);
 * -------------------------
 * This procedure adjusts the current point by moving it
 * dx inches from its current x coordinate and dy inches
 * from its current y coordinate.  As with MovePen, no
 * line is actually drawn.
 */

void AdjustPen(double dx, double dy)
{
    MovePen(GetCurrentX() + dx, GetCurrentY() + dy);
}


/*
 * Function: DrawLineTo
 * Usage: DrawLineTo(x, y);
 * ------------------------
 * This function is like DrawLine, except that it uses the
 * absolute coordinates of the endpoint rather than the relative
 * displacement from the current point.
 */
```

图7-6 函数AdjustPen和DrawLineTo

```
#define   DoorWidth           0.4
#define   DoorHeight          0.7
#define   DoorknobRadius      0.03
#define   DoorknobInset       0.07

#define   PaneHeight          0.25
#define   PaneWidth           0.2

#define   FirstFloorWindows   0.3
#define   SecondFloorWindows  1.25
```

这些值都是实数，表示英寸，名字描述了它们在图中的物理含义。在程序中，用符号化的名字表示这些值使得改变它们的大小更方便，例如，在你要使房子更宽一些，窗子更大一些时。

7.4.1 使用逐步精化

现在已经可以开始实施了。如第5章中所讨论，处理大的程序设计问题的最好方法是使用逐步精化策略，将整个问题分解成一组较小的问题。为了将此策略应用到画房子这个问题，可以从最综合的层次开始：要画一栋房子。要给这个工作取一个名字，如DrawHouse，并定义它为一个过程。实现DrawHouse过程成为你的第一个子问题。为了完成这个实现，你可以将整个问题分解成较小的部分：画轮廓、门和窗户。这些工作是分解过程中的下一层子问题。执行这个策略直到所有的子问题都简化为能用已有工具解决的最简单的操作。

然而，和DrawBox过程一样，你还要确定DrawHouse过程是否需要参数。在上一节中，已将房子的大小定义为常数。还要指出房子的位置，因此DrawHouse过程似乎取x和y两个坐标值是合适的，这两个坐标值指出了房子在图形窗口中的位置。为了与DrawBox过程一致， 241
这两个值指出房子的左下角的位置。因此DrawHouse过程的原型应该是

```
void DrawHouse (double x, double y);
```

调用此过程将指示计算机画一栋房子，它的左下角位于点（x，y）。

一旦定义了原型，现在就能开始来完成主程序。你需要做的是指出房子应该在屏幕上出现的位置。例如，假设你要把房子画在图形窗口的中间。如在本章前面的7.2.6节所讨论的，你可以用函数GetWindowWidth和GetWindowHeight找出窗口中心的坐标。例如，如果声明了变量cx和cy，可以用

```
cx=GetWindowWidth()/2;
cy=GetWindowHeight()/2;
```

设定中心的坐标。然而，一旦定义了DrawHouse，图形本身是相对于左下角而不是中心的位置画的。我们如何把这两个位置关联起来？

你已经知道该房子的宽为HouseWidth。因此房子的左面的边离中心的位置是这个距离的一半。如果房子左边的边的坐标为

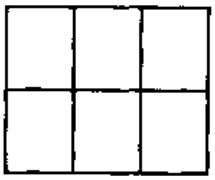
```
cx - HouseWidth/2
```

那么房子的中心是在屏幕的中心。你可以对y坐标重复同样的处理。唯一的差别在于房子的总高度是矩形和屋顶高度的和。因此房子左下角的y坐标为

```
cy - (HouseHeight + AtticHeight)/2
```

现在，你得到了房子左下角的坐标，就可以完成main的实现，程序如下：

为了设计适合于画窗的工具，尽可能考虑将这个问题通用化是不会错的。你的工具越通用，在各种情况下就越容易使用它们。保证你创建的工具能够广泛应用的一种方法是跳出当前问题的局限，而从一个更高的、更抽象的层次上认识所需要的操作。在房子这个问题上，右面的图描述了一个有几个窗格的窗。然而，当你集中于这个图本身时，你所见的是由两行组成的矩形网格，每行包括三个方格。如果有一个画一个矩形网格的过程DrawGrid，就可以用这个过程画每一组窗口。



DrawGrid需要什么参数？为了达到必要的通用性，必须保证DrawGrid过程不涉及房子问题的特性。因为用常量PaneWidth和PaneHeight会使过程只适用于本例房子的情况，最好是让调用程序将网格中的每一方框的宽度和高度作为参数来传递。调用程序知道这是画一扇窗，能为这个特定的应用提供PaneWidth和PaneHeight。这个过程本身只是画方框。除了网格中每个方框的高度和宽度外，DrawGrid也要知道完整的网格的坐标。为了与其他工具保持一致，DrawGrid将这些坐标解释为网格的左下角。最后，该过程必须知道网格中的列数和行数。因此DrawGrid的原型应该为：

```
Void DrawGrid(double x, double y, double width, double height,
              int columns, int rows);
```

假定已经有了函数DrawBox，DrawGrid的实现就非常简单。该实现由一对嵌套的for循环组成，它们在每一行中对每一列调用DrawBox，代码如下：

```
/*
 * Function: DrawGrid
 * Usage: DrawGrid(x, y, width, height, columns, rows);
 * ----------------------------------------------------
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid.  As always, (x, y) specifies the lower left corner
 * of the figure.
 */
void DrawGrid(double x, double y, double width, double height,
              int columns, int rows)
{
    int i, j;

    for (i = 0; i < columns; i++) {
        for (j = 0; j < rows; j++) {
            DrawBox(x + i * width, y + j * height,
                    width, height);
        }
    }
}
```

244

给定了DrawGrid的实现之后，可以通过用相应的参数调用DrawGrid来构造每扇窗口的模型。

尽管通常自顶向下设计问题是最好的方法，但自底向上实现问题通常也是最好的。先实现低层次的工具使它更易于调试程序的每个模块，这通常比一次性调试所有的程序更容易。这个策略称为**自底向上的实现**（bottom-up implementation）。

7.4.4 结束分解

给定了新的DrawGrid工具之后，画房子程序的剩下部分是一个逐步精化的最直接的例子。完整的程序house.c如图7-7所示。

```
                cy - (HouseHeight + AtticHeight) / 2);
}

/*
 * Function: DrawHouse
 * Usage: DrawHouse(x, y);
 * -----------------------
 * This function draws a house diagram with the lower left corner
 * at (x, y).  This level of the function merely divides up
 * the work.
 */

void DrawHouse(double x, double y)
{
    DrawOutline(x, y);
    DrawDoor(x + (HouseWidth - DoorWidth) / 2, y);
    DrawWindows(x, y);
}

/*
 * Function: DrawOutline
 * Usage: DrawOutline(x, y);
 * -------------------------
 * This function draws the outline for the house, using (x, y)
 * as the origin.  The outline consists of a box with a triangle
 * on top.
 */

void DrawOutline(double x, double y)
{
    DrawBox(x, y, HouseWidth, HouseHeight);
    DrawTriangle(x, y + HouseHeight, HouseWidth, AtticHeight);
}

/*
 * Function: DrawDoor
 * Usage: DrawDoor(x, y);
 * ----------------------
 * This function draws a door, with its doorknob.  As usual,
 * (x, y) specifies the lower left corner of the door.
 */

void DrawDoor(double x, double y)
{
    DrawBox(x, y, DoorWidth, DoorHeight);
    DrawCenteredCircle(x + DoorWidth - DoorknobInset,
                       y + DoorHeight / 2, DoorknobRadius);
}

/*
 * Function: DrawWindows
 * Usage: DrawWindows(x, y);
 * -------------------------
 * This function draws all the windows for the house,
 * taking advantage of the fact that the windows are all
 * arranged in two-dimensional grids of equal-sized panes.
```

246

图 7-7（续）

```
/*
 * Function: DrawCenteredCircle
 * Usage: DrawCenteredCircle(x, y, r);
 * -----------------------------------
 * This function draws a circle of radius r with its
 * center at (x, y).
 */

void DrawCenteredCircle(double x, double y, double r)
{
    MovePen(x + r, y);
    DrawArc(r, 0, 360);
}

/*
 * Function: DrawGrid
 * Usage: DrawGrid(x, y, width, height, columns, rows);
 * ----------------------------------------------------
 * DrawGrid draws rectangles arranged in a two-dimensional
 * grid.  As always, (x, y) specifies the lower left corner
 * of the figure.
 */

void DrawGrid(double x, double y, double width, double height,
              int columns, int rows)
{
    int i, j;

    for (i = 0; i < columns; i++) {
        for (j = 0; j < rows; j++) {
            DrawBox(x + i * width, y + j * height,
                    width, height);
        }
    }
}
```

249

图 7-7 (续)

小结

在本章中，我们开始研究接口的概念，接口是现代程序设计中最有用的想法。接口是库抽象的实现者和它的客户之间的连接点。接口指出了双方都需要知道的信息。在第8章，你将学到更多有关接口的知识，以及设计接口的策略。

本章介绍了一个特定的接口graphics.h，并将它作为介绍接口如何工作的通用实例。graphics.h接口使你可以在屏幕上的某一位置画线和弧等简单的图形。利用接口本身提供的函数，你也已经学会如何写一个其他的高级工具，如DrawBox、DrawCenteredCircle和DrawGrid，它们扩展了图形库的功能。

本章介绍的重点内容包括：

- 表示一个库的代码统称为*软件包*。软件包是某一抽象的程序表示，它是库的根本的概念基础。
- 库中的函数是由*实现者*编写，由*客户*调用。客户和实现者交汇的点称为*接口*。
- C语言中的接口用头文件表示。作为接口的头文件包含大量的文档以及由库导出的函数

```
DrawArc(4.0, -15, 2*15);
DrawArc(4.0, 180 - 15, 2 * 15);
```

16. 如何获取图形窗口中心的坐标?

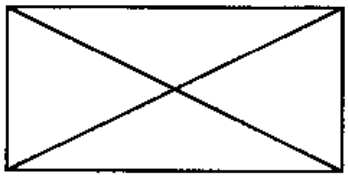
17. 建立类似于DrawBox和DrawCenteredCircle这样的新的过程有什么好处?

18. 当你设计一个作为工具的函数时，为什么跳出特定问题的范畴，更抽象地考虑该问题是非常有用的?

19. 自底向上的实现的意义是什么？优点是什么?

程序设计练习

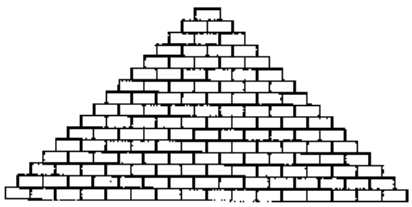
1. 写一个函数DrawCrossedBox，它的参数和DrawBox相同，但同时画上了矩形的对角线。例如，调用



```
DrawCrossedBox(0.5, 0.5, 1.0, 0.5);
```

将产生右面所示的图，该矩形的左下角在点(0.5, 0.5)。

2. 写一个程序，该程序画一个由一行行砖组成的金字塔，每向上移一层，行中砖的数目减1，如下图所示：



252

在你的实现中，应该用常量NBricksInBase指出最下面一行的砖数，用常量BrickWidth和BrickHeight指出砖的大小。

3. 画一个心型图案的一种方法是在一个旋转了45度后的正方形上面画两个半圆，如图7-8a所示。

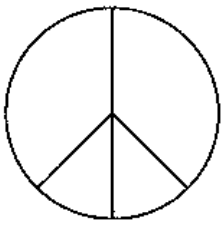
写一个程序，按这种结构在屏幕上画一颗心。该程序不要画出正方形上面的两条内部线，输出的图形看上去如图7-8b所示。



a) b)

图7-8 程序设计练习3的示意图

4. 1960年，下面这个符号



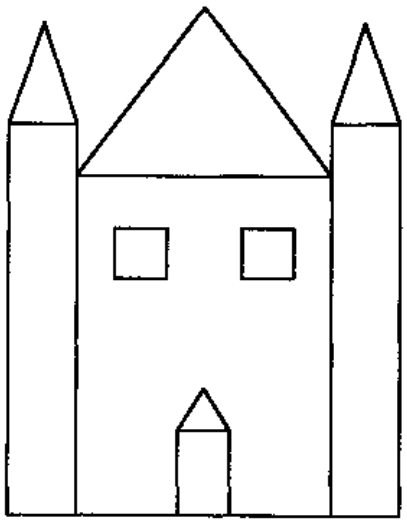
该能理解这些常量的意义。你的程序应该能做到修改这些常量会使图形有相应的变化。例如，如果将NTeethPerRow改为4，在新的图形中，每一行仅有四颗牙齿，但嘴仍然应该在水平位置的中间。南瓜脸的两个眼睛和嘴应该在两个方向上的中心到圆的边的中间位置上，因此改变HeadRadius也将改变这些特征的位置。表示南瓜的圆的中心应该是屏幕的中心。

8. 如果你要在练习7中设计的万圣节夜的南瓜旁画一栋房子，你可能要画一张Edgar Allen Poe描述的引座员的房子的图片，它的描述如下：

255

我看着前面的风景…寒冷的墙…空洞的像眼睛一样的窗…以及一个完全绝望的灵魂…

从Poe的描述中，你应该能想象出如右图所示的房子的样子。



写一个程序，画出如右图所示的房子，用下列常量指出各种尺寸：

```
#define HouseWidth          1.5
#define HouseHeight         2.0
#define HouseArch           1.0

#define TowerWidth          0.4
#define TowerHeight         2.3
#define TowerArch           0.6

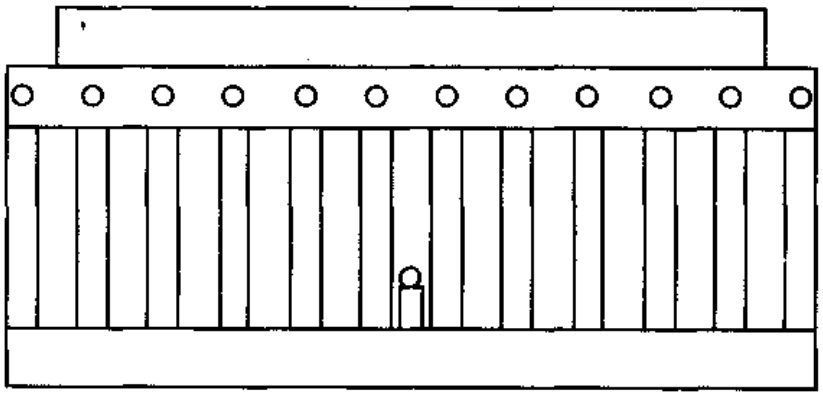
#define DoorWidth           0.3
#define DoorHeight          0.5
#define DoorArch            0.25

#define WindowLevel         1.4
#define WindowSize          0.3
```

以Arch结尾的常量指出矩形上面的三角形的高度。假设窗为正方形，因此只有一个尺寸WindowSize。

256

9. 写一个程序，画下面的位于华盛顿特区的林肯纪念馆的示意图：



和练习7和8一样，你的程序应该用下列常量定义图形的特征：

```
#define MemorialWidth       4.0
#define PedestalHeight      0.3

#define NumberOfColumns     12
#define ColumnWidth         0.15
#define ColumnHeight        1.0
```

第8章 设计接口：一个随机数库

And such a wall as I would have you think
That had in it a crannied hole, or chink
Through which the lovers, Pyramus and Thisbe
Did whisper often, very secretly

— Shakespeare, *A Midsummer Night's Dream*, 1595-1596

学习目标

- 了解评价接口设计的主要标准。
- 揭示程序如何利用伪随机数模拟随机行为。
- 理解库函数rand的行为。
- 学习如何使用算术运算修改伪随机数序列的范围。
- 认识接口项的常见类型。
- 学习写一个接口头文件的语法规则和习惯。
- 能够使用random.h接口提供的功能。

第7章已经介绍了接口的概念。此外，通过使用graphics.h接口，我们也了解了接口中有什么内容以及如何在程序设计中使用它们。但要完全理解接口，还必须学会如何实现它们。本章会让你设计一个新的接口以及它的基本实现。

按照所研究问题的范围，编写的接口可以是非常简单的，也可能是非常复杂的。如果仅考虑C语言的语法和结构，并没有许多新的规则要学。你已经学会了如何写注释和函数原型，它们是接口的主要组成部分。然而，一旦有了正确的算法，后面具有挑战性的工作并不是给这个接口编码，而是设计这个接口。重要的问题不是如何写一个接口，而是如何写一个好的接口。

设计一个好的接口是一个微妙的问题，需要平衡许多互相冲突的设计标准。本章将研究这些标准，说明它们的应用。为了使说明更加具体，本章将介绍一个库软件包开发的完整过程，该软件包提供了简单的随机数访问功能。

8.1 接口设计

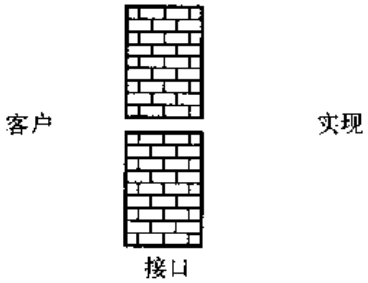
因为程序反映了它们所要解决的问题的复杂性，所以程序设计是非常困难的。只要你用计算机解决复杂性日益增长的问题，程序设计的过程也将变得更复杂。

在编写一个解决较大的或困难的问题的程序时，你必须处理大量的复杂性问题。其中有算法设计、要考虑的特殊情况、要满足用户的需求以及许多要保持正确的细节。要使得程序可管理，必须尽可能降低程序设计过程的复杂性。

在第5章中，我们已经学过使用函数和过程降低一些复杂性。接口提供了类似的作用，但它处于一个更高的细节层次上。函数让它的调用程序访问一组实现某一功能的一段程序。接口让它的客户访问一组实现某一程序设计中抽象行为的函数。接口对程序设计过程的简化程度在很大程度上依赖于它的设计。

为了设计一个有效的接口，必须在许多标准之间加以权衡。一般来讲，开发的接口应该

和实现之间互相通信。然而，墙的主要目的是保持两边分开。因为我们把它想象为库表示的抽象的边界，所以接口有时也被称为**抽象边界**（abstraction boundary）。理想情况下，库的实现中的所有的复杂性都位于墙的实现这一边。如果接口能对客户方屏蔽复杂性，那么接口就是成功的。把细节限制在实现范畴内称为**信息隐藏**（information hiding）。



信息隐藏的原理对接口的设计有重要的实际意义。当你写一个接口时，你应该确信没有显露出实现的细节，甚至在注释中也没有显露实现的细节。特别是，如果你同时写一个接口和一个实现，可能会不知不觉地在你的接口中记下你在写这个实现时的所有得意之处。一定不要这样做。写接口主要为了供客户使用，应该仅包含客户需要知道的内容。

同样，你应该在接口中设计这些函数，使它们尽可能简单。如果能减少参数个数或找到一种方法能消除特殊情况间的混乱，客户就容易理解如何使用这些函数。此外，限制接口输出 262
的函数个数通常是好的方法，这样客户不至于在大量的函数中迷失方向，找不到整体的意义。

8.1.3 满足客户的需要

任何事情都应该尽可能简单，而不是较简单。

——阿尔伯特·爱因斯坦

简单性只是接口设计原则的一个部分。抛弃接口中所有困难的或复杂的部分，就可以很容易地设计出一个接口。但这也使得一个接口变得毫无用处。有时，客户需要完成一些具有某种内在复杂性的任务。拒绝为客户提供他们所需的工具，可以使这个接口变得较简单，但这不是一个有效的策略。你的接口必须提供足够的功能以满足客户的需要。在接口设计中，学会在简单性和完备性之间达到合理的平衡是程序设计最基本的挑战之一。

在许多情况下，接口的客户不仅关心特定的函数是否可用，而且还关心实现的效率。例如，如果某一程序员正在开发一个航空控制系统，需要调用某一库接口提供的函数，这些函数必须迅速返回正确的结果。延迟返回的答案和错误的答案一样糟。

在大多数情况下，效率是实现而不是接口关心的因素。尽管如此，你还是经常会发现在设计接口本身时，考虑实现的策略是有价值的。例如，假设你有两个设计方案。如果能确定其中的一个能更容易更有效地实现，而另一个方案又没有绝对的竞争理由，选择这个设计是有意义的。

8.1.4 通用工具的优势

给我们工具，我们将完成工作。

——温斯顿·邱吉尔，广播演讲，1941

完全适合某一客户需要的接口对另外一些客户可能是毫无用处的。一个良好的库抽象要满足不同客户的需求。要做到这一点，它必须通用到足以解决很多问题，而不仅限于解决某种非常特殊的问题。通过选择一个能在客户使用这个抽象时具有足够灵活性的设计，就能创建一个应用范围很广的接口。

确保接口保持通用性有着重要的实用意义。当你写一个程序时，经常会发现需要一个特定的工具。如果你确定这个工具非常重要，值得放到一个库里，那么你需要改变你的思维方法。当你为库设计一个接口时，你必须忘记第一次使你想起要做这个工具的应用，而为大多 263

一些表示随机过程的机制，如抛硬币或掷骰子。模拟这种随机事件的程序称为**非确定性**（nondeterministic）程序。

8.2.2 随机数和伪随机数

因为早期的计算机主要用于数值应用，用计算机生成随机性的想法通常以生成特定范围内的**随机数**（random number）来表示。从理论上看，没有办法事先确定值的数是随机数，它将是一组等概率的值中的一个。例如，掷骰子生成1～6之间的随机值。如果骰子是公平的，那么没有办法预测将出现什么数。6个值出现的可能性是相等的。

虽然随机数的概念有直观的意义，但却是在计算机内部很难表示的概念。计算机根据一组内存中的指令序列工作，因此以一种确定的模式运行。如何遵循一组确定的规则产生一组不确定的结果？如果一个数是由一个确定的过程产生的，任何用户应该能够通过同一组规则工作并等待计算机的响应。 265

事实上，计算机确实能够用确定性的过程产生所需要的随机数。从理论上说，一旦用户能够遵循同一规则，并等待计算机的响应，那么这个策略就可以工作了，没有人会为此操心。在大多数实际应用中，这个数是否为真正随机是无所谓的；真正在乎的是这个数是随机出现的。对随机出现的数，它们的行为应该像统计学中的随机数，而且提前预测它们非常困难，但没有用户为此操心。由计算机内部的算法产生的“随机”数称为**伪随机数**（pseudo-random number），强调没有涉及真正的随机活动。

8.2.3 ANSI C中生成伪随机数

作为stdlib.h接口的一部分，ANSI C库包含了一个产生伪随机数的函数rand。在接口中给出的rand原型是

```
int rand(void);
```

它指出rand没有参数，并返回一个整数，这就是一个伪随机值，即每次调用rand返回的结果都不同。rand的结果保证是非负的，并且不大于常数RAND_MAX，该常数也是在stdlib.h接口中定义的。因此，每次调用rand时，它返回的是0～RAND_MAX的不同的整数，包括0和RAND_MAX。

RAND_MAX的值取决于计算机系统。在典型的Macintosh环境中，RAND_MAX的值是32 767。在典型的Unix工作站上，它的值是2 147 483 647。当你写一个用到随机数的程序时，不需要对RAND_MAX的值作精确的假设。相反，你的程序应该能在任意的RAND_MAX的系统上使用。如果你做到了这一点，你可以将一个在某一系统上工作的程序重新编译一下，它就能在另一个系统上工作。

运行图8-1中的程序randtest.c，将会显示rand的行为。

在我的一台计算机上，randtest.c产生了下列输出：

```
On this computer, RAND_MAX = 32767.
Here are the results of 10 calls to rand:
      16838
       5758
      10113
      17515
      31051
       5627
      23010
       7419
      16212
       4086
```

266

用这个想法可以开发出图8-2所示的cointest.c程序，它可以模拟抛硬币的过程。

```
/*
 * File: cointest.c
 * ----------------
 * This program simulates flipping a coin.
 */

#include <stdio.h>
#include <stdlib.h>
#include "genlib.h"

/*
 * Constants
 * ---------
 * NTrials -- Number of trials
 */

#define NTrials 10

/* Main program */

main()
{
    int i;

    for (i = 0; i < NTrials; i++) {
        if (rand() <= RAND_MAX / 2) {
            printf("Heads\n");
        } else {
            printf("Tails\n");
        }
    }
}
```

图8-2 cointest.c

该程序打印出字符串“Heads”或字符串“Tails”，每种输出的出现几率大约为50%。如果测试这个程序，可以得到如下的运行示例：

```
Tails
Heads
Heads
Tails
Tails
Heads
Tails
Heads
Heads
Heads
```

正面和反面有合理的组合，你不能很容易地找出它的模式。

在考虑如何将rand的结果转换为两种可能时，许多新的程序员最初可能对一些比较简单的方法感兴趣，如用取余数的运算。如果将rand的结果除以2，并取它的余数，结果是0或者

• 否则，如果产生的随机数小于最大值的3/6，返回3，以此类推。

问题是，你的代码没有完全实现这个想法。通过重复调用函数rand，在每一个if语句中都将产生一个新的随机数。该函数的结构依赖于每次产生的随机数都相同的假设。要了解这种情况，观察一下这个RollDie错误百出的实现，试着了解一下在什么条件下它将会返回2。为了使函数返回2，第一个if语句必须为FALSE，第二个语句必须为TRUE。第一个if语句的条件为FALSE的概率为5/6。第二个if语句会使得返回TRUE的概率为1/3，因为调用rand返回的是一个全新的随机数。在统计学上，两个独立事件一起出现的概率为两个独立事件出现的概率的乘积，因此RollDie返回2的概率是

$$\frac{5}{6}\times\frac{1}{3}=\frac{5}{18}$$

18次中出现5次的机会几乎是6次中出现1次的机会的2倍，这意味着RollDie函数返回2的机会比应该出现的机会大。为了解决这个问题，需要声明一个变量保存rand的结果，以后在每行中测试这个变量，如图8-4所示。

```
int RollDie(void)
{
    int r;

    r = rand();
    if (r < RAND_MAX / 6) {
        return (1);
    } else if (r < RAND_MAX * 2 / 6) {
        return (2);
    } else if (r < RAND_MAX * 3 / 6) {
        return (3);
    } else if (r < RAND_MAX * 4 / 6) {
        return (4);
    } else if (r < RAND_MAX * 5 / 6) {
        return (5);
    } else {
        return (6);
    }
}
```

(该实现仍然不正确)

图8-4 RollDie的第二次尝试

常见错误 当调用一个产生伪随机数的函数时，重要的是要记住每次调用这个函数都会产生一个不同值。如果要保存一个特定值，必须把函数的结果保存在一个变量中。

271

但是，这个实现还是错误百出。然而，第二个问题更加不明显了。在大多数系统上，RAND_MAX的值是按某种原因给出的。通常为RAND_MAX选择的值不一定是rand函数可能返回的最大值，而是系统中int类型的最大的正值。这个限制在所建议的RollDie的实现中引起了一个严重的问题，因为该程序中的某些中间结果可能大于最大的整数值。尽管

```
RAND_MAX * 2/6
```

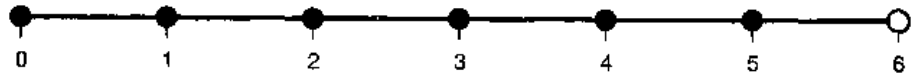
最终的结果适合int类型，但C语言中的优先规则指出RAND_MAX先乘2，再除6。产生一个所允许范围之外的整数称为**运算溢出**（arithmetic overflow）。如果出现这样的溢出，程序将不能产生预期的结果。你可以把表达式改为

严格小于1。因此，在这个过程结束时，可以得到一个至少为0，但严格小于1的随机实数。在数学中，可以等于一个端点值，但不能等于另一个端点值的实数范围称为**半开区间**（half-open interval）。在图中，不包括在范围内的端点用空心圆表示。因此变量d的可能的范围可以用下图表示：



273

下一步是换算这个值，使它覆盖正确的整型数的数目。例如，要模拟掷骰子，需要把这个值乘6，因此新的范围为



注意，在这个范围中有6个整数：整数0、1、2、3、4和5。6本身在可能值的范围外，因为d永远不可能为1。

在更一般的情况下，要将规范化的随机数乘以这个范围内的元素个数，元素个数可以用表达式

```
(high - low + 1)
```

表示。表达式中多出来的这个1是必须的，因为该范围是包括两个端点的。减法给出了两点之间的距离，它比闭区间中包含的整数少1。对掷骰子来讲，可能出现六个值是1、2、3、4、5和6，但6减1是5。为了计算出现的值的个数，需要将最大的数减去最小的数，然后加1。

下一步，向后截断该实数，使它成为一个整数。在C语言中，如果用强制类型转换将double转换为int，那么该转换是通过丢掉小数部分实现的。因此，如果要将一个已知的实型数转换为大于等于0，但严格小于6的数，将得到0、1、2、3、4和5中的某一个整数。

过程中的最后一步是将结果转换到所期望的范围内。已经有了整型结果的正确个数；仅有的问题是它们是从0开始。为了得到一组正确的可能性，只要简单地加上下界值即可。

可以把所有的这些步骤放在一起，写出如下的RandomInteger函数的实现：

```
int RandomInteger(int low, int high)
{
    int k;
    double d;

    d = (double) rand() / ((double) RAND_MAX + 1);
    k = (int) (d * (high - low + 1));
    return (low + k);
}
```

8.3 在库中保存工具

RandomInteger函数十分有用，因此可以把它放入某一个库中。这个过程的第一步是建立一个接口。一旦完成了这个接口，就可以在另一个文件中写出对应的实现。在大多数情况下，除了文件类型外，接口和它的实现的文件名是相同的。因此，如果接口被命名为random.h，一般会用random.c作为实现文件的名字。

274

8.3.1 接口的内容

接口的基本结构在第7章介绍的graphics.h实例中已作了说明。和本书中引入的所有接口一样，graphics.h主要由为使用该库的客户所编写的注释组成。这些注释是接口的关键

除了模板文件外，接口有时也需要用同样的#include行包含其他已经在程序中使用的接口。当用到这样的行时，所用的规则将在本章的8.4.3节讨论。

8.3.2 写random.h接口

如果将上一节讨论的规则用于编写random.h接口，你会认识到你的第一个任务是写初始的注释，这些注释将解释这个库提供了什么，谁可以使用它。在注释后面，应该包括接口的模板文件，本例中的模板文件是

```
#ifndef _random_h
#define _random_h
```

在random.h接口中，下一个要写的是有关RandomInteger过程的注释：

```
/*
 * Function: RandomInteger
 * Usage: n = RandomInteger(low, high);
 * ------------------------------------
 * This function returns a random integer in the range
 * low to high, inclusive.
 */
```

这段注释给客户提供使用这个函数所需的信息。例如，Usage这一行给出了一个调用函数的实例，对客户来说特别有帮助。这段注释也包括这个函数功能的说明，但没有说明函数是如何实现的。

接口的下一部分是函数本身的原型：

```
int RandomInteger(int low, int high);
```

这一行是接口中仅有的对编译器有实际意义的部分；其他都是注释或模板文件。

接口中的最后一行是简单的#endif行，它是接口的模板文件的一部分。 276

至今为止讨论过的random.h接口中的部分可以参见图8-5。如它的标题所示，图8-5中的接口仅是最初的版本。在本章后面，还会加入新的函数，以扩展它的功能。

```
/*
 * File: random.h
 * --------------
 * This file contains a preliminary version of a library
 * interface to produce pseudo-random numbers.
 */

#ifndef _random_h
#define _random_h

/*
 * Function: RandomInteger
 * Usage: n = RandomInteger(low, high);
 * ------------------------------------
 * This function returns a random integer in the range
 * low to high, inclusive.
 */

int RandomInteger(int low, int high);

#endif
```

图8-5 random.h的最初版本

8.3.4 构造客户程序

你可以写一个如图8-7所示的程序dicetest.c，来测试random.c的实现。主程序使你可以使用新的随机函数库，因此在dicetest.c文件中需要包括行

```
#include "random.h"
```

277
↓
278

使它能用RandomInteger函数。

```
/*
 * File: dicetest.c
 * ----------------
 * This program simulates rolling a die.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"

/*
 * Constants
 * ---------
 * NTrials -- Number of trials
 */

#define NTrials 10

/* Function prototypes */

int RollDie(void);

/* Main program */

main()
{
    int i;

    for (i = 0; i < NTrials; i++) {
        printf("%d\n", RollDie());
    }
}

/*
 * Function: RollDie
 * Usage: die = RollDie();
 * -----------------------
 * This function generates and returns a random integer in the
 * range 1 to 6, representing the roll of a six-sided die.
 */

int RollDie(void)
{
    return (RandomInteger(1, 6));
}
```

图8-7 dicetest.c

279

让我们迅速测试一下这个程序，确保它能工作。运行此程序的结果如下：

每次都返回同一序列值。

尽管rand的系统定义有利于调试，但一旦程序开始工作仍能够改变它的行为也是很重要的。然而，了解如何进行改变需要知道一些rand的实现的知识。

ANSI库通过记录最近生成的数产生伪随机数。每次调用rand时，它取最近生成的一个数，对这个数执行一系列的操作，产生下一个数。因为你不知道到底做了哪些运算，最好把整个操作看成一个黑盒子，旧的数从盒子的一边进入，新的伪随机数从另一边弹出。

在本章前面的8.2.3节中描述的randtest.c程序提供了rand内部操作的说明。在我的计算机上，前十次调用rand生成的数如下所示：

```
On this computer, RAND_MAX = 32767.
Here are the results of 10 calls to rand:
     16838
      5758
     10113
     17515
     31051
      5627
     23010
      7419
     16212
      4086
```

281

第一次调用rand产生16838。第二次调用对应于将16838放入表示内部实现的黑盒的一端，使5758从另一边弹出来，见右图。



同样，下一次调用rand时，该实现将5758放入黑盒子中，返回10113，见下图。在每次调用rand时，都重复同样的过程。黑盒内部的计算要满足以下条件：(1) 数值在合法的范围内均匀分布，(2) 该序列能持续很长时间才开始重复。



但第一次调用rand（返回16838的那次）的过程又是怎样呢？该实现必须有一个开始点。必须有一个整型数s被输入到黑盒子，以产生16838，见右图。这个用来使整个过程开始的初始值被称为随机数生成器的**种子**（seed）。每次程序开始时，ANSI的库实现将初始种子设为一个常量，因此总是产生同一序列值。将种子设为不同的值就能改变这个序列。要做到这一点，需要调用函数srand，它将一个新的种子作为参数。为了保证新的种子值在每次程序运行时都变化，标准的做法是用内部系统时钟值作为初始种子。因为时间一直在变，所以随机数序列也将变化。



可以调用函数time获取系统时钟的当前值，该函数在ANSI库接口time.h中定义，然后将返回值转换为整型数。这个技术允许你利用下列语句将伪随机数生成器初始化为某一个不可预测点：

```
srand((int) time(NULL));
```

虽然它只有一行，但基于系统时钟将随机种子设为一个不可预测值的操作是相对模糊的。如果这一行出现在客户程序中，客户必须理解随机数种子的概念、time函数以及神秘的常量

282

再定义一个函数以便提供所需的功能，与RandomInterger相对应，可以称为RandomReal。

8.4.1 产生随机实数

碰巧，作为RandomInteger实现的一部分，你已经使用rand函数来产生随机实数。生成随机整数过程的第一步就是生成一个0～1之间的随机浮点数。要实现RandomReal函数，一种方法是做同样的计算，返回结果。然而，这样的设计在某种程度上违反了保证库一致性的同一性原则。用这种方法，RandomReal没有参数，并返回一个处于预先设定范围内的浮点值。而RandomInteger函数的行为则不同。它有两个参数，并返回由输入定义的范围内的值。为保证一致性，最好让RandomReal有同样的设计。如果这样做的话，了解RandomInteger是如何工作的客户就能正确预测RandomReal的结构。因此，RandomReal应该有如下所示的原型：

```
double RandomReal(double low, double high);
```

它的实现基本上是由RandomInteger实现的前两行组成，只是换算因子是确切的两个端点之间的距离而不是包含在此范围内的整型数个数。因此，RandomReal的实现为

```
double RandomReal(double low, double high)
{
    double d;

    d = (double) rand() / ((double) RAND_MAX + 1);
    return (low + d * (high - low));
}
```

284

8.4.2 模拟一个概率事件

除了随机实数以外，在模拟随机行为的一般抽象中，还有另一种有用的随机变量的类型。假设你要写一个程序，其中需要一个事件以某种随机概率出现。例如，假设你的程序要模拟一条装配线，这条装配线平均每传送1000个零件就要出一次故障。用仿真的术语来讲，是每个零件有千分之一的故障率。在数学和统计学中，概率是0～1之间的数，因此，这个例子中故障率是0.001（1/1000）。

在这个例子中，结果只有两种可能：有故障或者没有故障。表示某个条件出现或不出现的只有两个结果的事实用布尔值表示是最恰当的。用TRUE表示检测到故障，它出现的概率是0.001。

在这种情况下，有一个以某种概率返回TRUE的谓词函数是很有用的。如果你已经用了这样的函数，它的名字可能为RandomChance，则可以用下列的代码表示这条装配线：

```
if (RandomChance(.001)) {
    printf("A defect has occurred.\n");
}
```

作为第二个实例，你也可以用RandomChance模拟投硬币：

```
if (RandomChance(.5)) {
    printf("heads\n");
} else {
    printf("tails\n");
}
```

与本章前面的8.2.4节中的实现方法相比，这种实现的优点在于它不需要客户理解rand函数本身的操作。客户仅依赖于更高级的random.h接口中定义的函数。rand函数的存在仅仅被认

子的结果，分解出下列几种情况：

- 掷出2、3或12，第一次掷骰子得到这些结果称为放弃，意味着你输了。
- 掷出7或11，第一次掷骰子得到这些结果称为自然赢，表示你赢了。
- 如果是其他数（4，5，6，8，9或10）。在这些情况下，掷到的数称为你的点，你继续掷骰子直到第二次掷到你的点或者是7，如果掷到你的点，你赢了；如果掷到7，你输了。如果掷到其他数（包括2，3，11和12，这些不再特殊处理），继续掷骰子直到出现你的点或7。

```
/*
 * File: craps.c
 * -------------
 * This program plays the dice game called craps.  For a discussion
 * of the rules of craps, please see the GiveInstructions function.
 */

#include <stdio.h>
#include "genlib.h"
#include "random.h"
#include "simpio.h"
#include "strlib.h"

/* Function prototypes */

void GiveInstructions(void);
void PlayCrapsGame(void);
int RollTwoDice(void);
bool GetYesOrNo(string prompt);

/* Main program */

main()
{
    Randomize();
    if (GetYesOrNo("Would you like instructions? ")) {
        GiveInstructions();
    }
    while (TRUE) {
        PlayCrapsGame();
        if (!GetYesOrNo("Would you like to play again? ")) break;
    }
}

/*
 * Function: GiveInstructions
 * Usage: GiveInstructions();
 * --------------------------
 * This function welcomes the player to the game and gives
 * instructions on the rules to craps.
 */

void GiveInstructions(void)
{
```

图8-8 craps.c

```
    printf("Rolling the dice . . .\n");
    d1 = RandomInteger(1, 6);
    d2 = RandomInteger(1, 6);
    total = d1 + d2;
    printf("You rolled %d and %d -- that's %d.\n", d1, d2, total);
    return (total);
}

/*
 * Function: GetYesOrNo
 * Usage: if (GetYesOrNo(prompt)) . . .
 * ------------------------------------
 * This function asks the user the question indicated by prompt
 * and waits for a yes/no response.  If the user answers "yes"
 * or "no", the program returns TRUE or FALSE accordingly.
 * If the user gives any other response, the program asks
 * the question again.
 */

bool GetYesOrNo(string prompt)
{
    string answer;

    while (TRUE) {
        printf("%s", prompt);
        answer = GetLine();
        if (StringEqual(answer, "yes")) return (TRUE);
        if (StringEqual(answer, "no")) return (FALSE);
        printf("Please answer yes or no.\n");
    }
}
```

图 8-8 (续)

287 ~ 289

程序本身是将自然语言描述的规则直接转换为C代码。当你通读craps.c程序时，应注意下列特征：

- 该程序包括接口random.h，因此，它能使用这个库中的函数。此外，这个程序仅应用那个库中的函数，从来没有直接调用rand（或srand、time）。随机数序列是调用Randomize来初始化的，每次掷骰子是调用RandomInteger产生的。
- 程序被分解成更小的单元，这些单元给出更详细的信息。这种分解有助于突出程序的结构，使你能理解这些单元是如何组合起来的。
- 掷两个骰子的问题出现在程序中的多个地方，因此模拟掷两个骰子、显示结果和记住总和这组动作被封装在一个函数RollTwoDice中，它也能被用在其他的地方。

小结

在本章中，你体验了编写一个接口和它对应的实现的过程。在某种层次上，你已经了解接口的语法结构和它包含的内容。也已经学习了一些接口设计的最一般的原则，即在解决一些大问题时非常重要一些原则。最后，还看到了在构造random.h接口时如何应用这些原则。

本章介绍的重点内容包括：

- 构造一个接口的挑战在于接口的设计而不是编码。
- 一个定义良好的接口必须具备同一性、简单性、充分性、通用性和稳定性。有时这些标

模拟掷两个骰子的过程？

11. 三个最常见的接口项是什么？

12. 如果定义一个名为magic.h的接口，接口的模板是什么样的？这些行的作用是什么？

13. **判断题**：每次程序运行时，rand函数一般生成的是同一个随机数序列。

14. 在随机数中，术语*种子*的含义是什么？

15. 调试包含随机数的程序，本章提供了哪些建议？

16. 在random.h接口的最终版本中，定义了哪些函数？这些函数在哪些场合下使用？

程序设计练习

1. 在你的计算机系统上运行randtest.c程序，计算机上的RAND_MAX的值是什么？

2. 写一个显示2～100之间的偶随机数的程序。

3. 写一个随机显示7位电话号码的程序。输出应该服从在美国应用的以下规则：

- 输出的3～4位之间有一个破折号，如555-1968。
- 前两位都不能是0或1。这条规则实际上在美国的许多地方都已不采用，不过在这个问题中你还应该使用这条规则。因此，你的程序可能产生781-9902这样的电话号码，而不会产生718-9902这样的电话号码。 292

4. 写一个程序显示从52张扑克牌中随机抽取到的一张牌的名字。每张牌有它的等级（A, 2, 3, 4, 5, 6, 7, 8, 9, J, Q, K）和花色（梅花，方片，红桃，黑桃）。程序应显示完整的牌的名字，如下列运行示例所示：

```
Queen of Spades
```

5. **Heads. . . .**
Heads. . . .
Heads. . . .
A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.

—*Rosencrantz and Guildenstern Are Dead*, Tom Stoppard, 1967

写一个程序模拟反复投硬币，直到连续出现三个正面为止的情况。此时，你的程序应该显示投掷硬币的总次数。下面是程序可能的运行结果：

```
tails
heads
heads
tails
tails
heads
tails
heads
heads
heads
It took 10 flips to get heads 3 consecutive times.
```

```
 2        2464
 3        1207
 4         627
 5         311
 6         166
 7          89
 8          40
 9          21
10           8
11           4
12           1
13           0
```

正如这些数字所指出的，每年在样本中衰减了约一半的原子。在物理中，表示这个观察结果的常用说法是这个样本每年有一个半衰期。

295

8. 随着计算机在学校中越来越普及，找到如何在教学过程中用好计算机的方法是很重要的。这促进了教学软件业的发展，而且生产了许多帮助教学的工具。

作为教学应用的一个实例，写一个程序，它能产生一系列简单的算术问题让学生回答，如下所示：

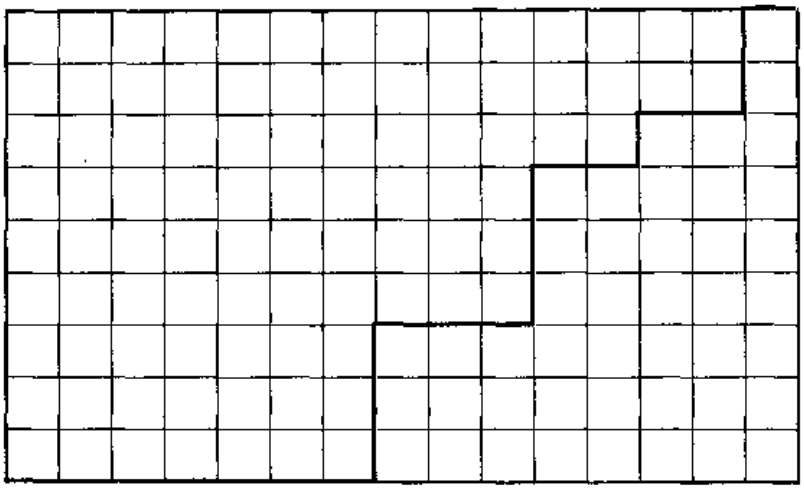
```
Welcome to Math Quiz!
What is 14 + 2? 16↵
That's the answer!
What is 17 - 15? 17↵
That's incorrect.  Try a different answer: 15↵
That's incorrect.  Try a different answer: 3↵
No, the answer is 2.
What is 20 - 16? 4↵
That's the answer!
What is 9 + 4? 11↵
That's incorrect.  Try a different answer: 13↵
That's the answer!
What is 11 - 1? 10↵
That's the answer!
```

你的程序应该满足以下需求：

- 它可以连续问五个问题。因为有这样的限制，所以问题数应该用#define常量定义，使之容易修改。
- 每个问题应该是仅涉及两个数的简单的加减问题，如“2＋3等于多少？”或“11－7等于多少？”。每一个问题的类型（即加或减）应该是随机选择的。
- 为了保证问题适合一、二年级的学生，任何数字（包括答案）都不能小于1或大于20。这个限制意味着程序不应该问类似于“11＋13是多少？”或“4－7是多少？”的问题，因为答案超出了合法的范围。在这些约束下，你的程序应该随机选择数字。
- 对每个问题，程序应该给学生三次回答的机会。如果学生给出了正确的答案，程序应该能以某种方式表示祝贺，并继续下一问题。如果学生三次都没能给出正确的回答，程序应该给出答案，并继续另一个问题。

296

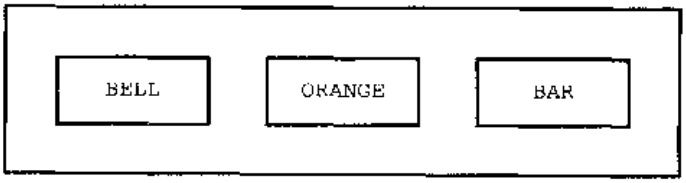
9. 尽管练习8中的程序在学生正确回答问题后会给出鼓励。但是只反复显示如“对了！”



写一个程序，用图形库描绘出通过城市的一条随机路线。开始时，将笔移到右上角。从这里开始，以随机的方式选择方向，画一条水平的或垂直的线到下一个交叉点。继续这个过程，直到你到达家中，但确保不要走到地图的外面去。 298

该程序不需要画一张完整的地图，只要直接显示这条路线就足够了。然而，如果想要更熟练地使用图形库，应该画出例子中所示的随机路线的完整的图。为了用粗线条表示随机路线，可以在实际格子线的每一边画一根直线。

12. 有一种叫做“碰运气”的游戏工具[⊖]。它有三个轮子，在一个很窄的窗口后面旋转。每个轮子上有下列符号标记：CHERRY、LEMON、ORANGE、PLUM、BELL和BAR。然而，在某一时刻你只能看到每个轮子上的一个符号。例如，窗口可能显示以下的配置：



开始游戏时，轮子开始旋转，最终出现某一个新的配置。如果该配置与规则规定的某个赢的组合相匹配，你就赢了。否则，你就输了，会被扣一分。下表显示了一个典型的赢模式的集合以及相应的得分：

BAR	BAR	BAR	得分	250
BELL	BELL	BELL/BAR	得分	20
PLUM	PLUM	PLUM/BAR	得分	14
ORANGE	ORANGE	ORANGE/BAR	得分	10
CHERRY	CHERRY	CHERRY	得分	7
CHERRY	CHERRY	—	得分	5
CHERRY	—	—	得分	2

符号BELL/BAR指的是在这个位置出现BELL或BAR，破折号指的是这里可以出现任何符号。因此，在第一个位置是CHERRY，不管其他轮子出现了什么，自动得到2分。注意，符号LEMON不会有得分，甚至碰巧三个都是LEMON时也一样。

写一个程序，模拟这个游戏。该程序给用户50分的初始分，让用户玩，直到分被扣完或

⊖ 本题在英文原书基础上有所改编。——编辑注

第9章 字符串和字符

Surely you don't think numbers are as important as words.

— King Azaz to the Mathemagician, *The Phantom Tollbooth*, Norton Juster, 1961.

学习目标

- 理解枚举的原理和用整数编码作为定义新的数据类型的策略。
- 能定义和操作C语言中的枚举类型。
- 理解计算机系统如何表示数据类型char，如何操作char类型的对象。
- 学习如何使用ctype.h接口中的函数。
- 理解抽象类型的概念。
- 能用strlib.h接口中的函数实现字符串操作。

301

到目前为止，你在本书中看见的绝大多数的程序设计的例子都是用数字作为基本的数据类型。正如Juster在Mathemagician中所坚持的，数字确实很重要，但在世界上还有许多其他类型的数据。目前，计算机较少用于数值型的数据，而较多用于**文本数据**（text data），即由键盘和屏幕上出现的独立的字符组成的信息。现代计算机处理文本数据的能力推动了字处理系统、在线参考库、电子邮件以及其他许多非常有用的应用的发展。

文本数据的概念在第2章中已正式引入，以本书的第一行代码为例，它包括字符串

```
"Hello, world.\n"
```

从那时开始，你已经见过如何用GetLine读一个字符串的值，如何用printf显示字符串，如何用StringEqual比较两个字符串。然而，这些操作仅仅是字符串操作中的很少的一部分。为了发挥文本数据的所有能力，需要知道如何以更复杂的方法操作字符串。本章中，你将学习如何使用字符串操作的库，它将帮助你写出更有创造性、更有趣的包含文本数据的程序。

因为字符串是由独立的字符组成，因此了解字符是如何工作的，以及它们如何在计算机内部表示是非常重要的。因此，本章重点介绍数据类型char以及数据类型string。然而，在研究这些类型的细节之前，需要从更一般的观点了解数据的表示。

9.1 枚举的原理

随着计算机技术的发展，越来越多的信息要以电子化的方式存储。为了将信息存储在计算机中，需要将数据表示为机器能使用的形式。特定内容的表示取决于它的数据类型。整数在计算机内有一种表示方式，而浮点数有另一种表示方式。尽管你不知道这些数据究竟是怎样表示的，但是这取决于计算机能在它的内部存储器中存储数据这个事实。除了数字之外，还有许多有用的数据类型，因此计算机必须也能表示这些非数值型的数据。

为了洞察非数值型数据的特性，考虑一下你自己在一年中提供给学校和机关的信息。例如，如果你生活在美国，你要将你的年退税数据提供给国内税收服务部。这些信息绝大部分是数值型的，如你的工资、扣除额、税、代扣所得税等。有些是由文本数据组成的，如你的名字、地址、职务。但还有一些项目不能简单归入这两类。例如，问题之一是

302

```
#define Tuesday      2
#define Wednesday    3
#define Thursday     4
#define Friday       5
#define Saturday     6
```

为了用这种方法表示枚举类型，必须显式指定整数编码。如果要引入一个变量保存这种类型的值，可以把它定义为int型。

也可以用这种方法表示退税单上的报税身份，可以定义如下常量：

```
#define Single                          1
#define MarriedFilingJointReturn        2
#define MarriedFilingSeparateReturn     3
#define HeadOfHousehold                 4
#define QualifyingSurvivingSpouse       5
```

最后一个常量的名字已经从税单上的名字改为符合C语言规则的名字，因为C语言的名字中不允许出现圆括号。

一旦定义了这些常量，就可以声明一个int类型的变量来表示报税身份：

```
int filingStatus;
```

9.1.3 定义新的枚举类型

尽管上一节中描述的方法在C程序中很常见，但它并没有完全发挥该语言提供的功能的所有优势。如果不使用int类型表示非数字值，在C语言中，可以定义一个真正的类型名去表示一种枚举类型。在机器内部，这两种策略产生的结果完全相同：枚举类型中的每个元素都用一个整数代码表示。从程序员的观点来看，定义单独的枚举类型有三个优势： 304

- 编译器能选择整数代码，从而将程序员解脱出来。
- 有一个独立的类型名使程序容易读，因为声明时可以使用有意义的名字，而不是通用的符号int。
- 在许多计算机系统中，使用显式定义的枚举类型的程序容易调试，因为编译器可以为调试系统提供有关该类型行为的额外信息。

C语言提供了多种定义新的枚举类型的语法形式。在本书中，所有新的枚举类型都是用如下所示的语法定义的。该定义以关键词typedef开头，它表示引入了一种新的类型名。在这里，关键词typedef后面紧跟着关键词enum，它指出新的类型是枚举类型。然后，枚举类型中的每个元素的名字列在一对花括号中，再接下去是类型的名字和一个分号。（关键词typedef也可以用在其他类型定义中，这将在第16章讨论。）

语法：枚举类型的定义

```
typedef enum {
    list of elements
} type name;
```

其中：

*list of elements*是由组成枚举类型的每个值的名字组成的。表中的元素用逗号分开。每个元素后也可以紧跟一个等号和一个整数常量，它指出了特定的内部表示。

*type name*指出新的枚举类型的名字。

定义指南针上的四个基本方向。每个这样的定义都引入了一个新的类型名和对应于该类型元素的一组常量。

事实上，genlib.h中的bool类型的定义就是一个枚举类型：[⊖]

```
typedef enum {FALSE, TRUE} bool;
```

9.1.4 枚举类型的操作

一旦在表达式中用到枚举类型的值时，C编译器自动将它转换为整数。因为每个枚举常量都用一个整数值表示，所有涉及枚举类型的计算只是用到它的整型代码代替。例如，如果你声明一个变量weekday为weekdayT类型，你仍然可以用下面的表达式

```
weekday = (weekday + 1) % 7;
```

就和在calendar.c程序中一样。变量weekday可以是0～6之间的某一个内部整数值。该语句将此整数值加1，然后通过模7取余操作保证结果还在0～6之间。枚举类型的所有算术运算都与整数一样。

然而，在C语言中，使用枚举类型时要非常小心，因为编译器不检查计算的结果是否是某一枚举类型的合法成员。例如，如果你误写了一个语句

```
weekday = weekday + 1;
```

如果weekday的值碰巧为Saturday，就会遇到麻烦。计算机将取表示Saturday的值6，对它加1，把这个值7存回到变量weekday，尽管7不是weekdayT类型的合法元素。C编译器应该生成一段额外的代码去检查这个条件，并报告错误，但很少有编译器这样做。当然，如果你将weekday作为一个整数，不考虑取余数的操作也是会产生错误的，因此，选择使用枚举类型不是问题的起因。

9.1.5 标量类型

行为类似于整数的类型，如枚举类型，称为**标量类型**（scalar type）。在C语言中，当你在表达式中使用标量类型时，它们被自动转换为整数。此外，标量类型可以用在任何整数类型出现的地方。例如，一个枚举类型的变量能被用在switch语句的控制表达式中。假设directionT被定义为 307

```
typedef enum {North, East, South, West} directionT;
```

下面的函数返回参数指定的方向的反方向：

```
directionT OppositeDirection(directionT dir)
{
    switch (dir) {
      case North: return (South);
      case East:  return (West);
      case South: return (North);
      case West:  return (East);
      default:    Error("Illegal direction value");
    }
}
```

⊖ 在一个表示布尔条件的C语言表达式中，整数0表示条件为假，尽管整数1是常量TRUE的“官方”值，但实际上任何非0的值都被解释为真。C程序员经常利用C语言解释非0整数为真的条件，尽管本书中的程序没有这样做。限制你自己在条件中使用明确的布尔值，将有利于你逐渐养成良好的程序设计习惯，这将使得你的程序比较容易读和调试。

即65。以同样的方法，能找出任何字符的编码。然而，在大多数情况下，并不需要这样做。虽然知道字符在计算机内部用什么数字编码很重要，但知道哪个数字值对应于哪个特定字符并不是很有用。当你键入字母A时，键盘中的硬件自动将此字符翻译成ASCII代码65，然后把它发送给计算机。同样，当计算机把ASCII代码65发送到屏幕时，屏幕上会出现字母A。

表9-1 ASCII代码

	0	1	2	3	4	5	6	7	8	9
0	\000	\001	\002	\003	\004	\005	\006	\a	\b	\t
10	\n	\v	\f	\r	\016	\017	\020	\021	\022	\023
20	\024	\025	\026	\027	\030	\031	\032	\033	\034	\035
30	\036	\037	空格	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	\|	}	~	\177		

9.2.3 字符常量

当你在一个C语言程序中用到某一特定字符时，标准的方法是指定一个**字符常量**（character constant），即将所需的字符括在一对单引号中。例如，要指出字母A的ASCII值，所需要做的只是写'A'。C编译器知道这个符号指的是用字母A的ASCII值，即65。同样，可以用' '指出空格，或用'9'指出数字9。注意，常量'9'指的是一个字符，不应该与整数值9混淆。作为一个整数，值'9'是ASCII表中给出的这个字符的值，即57。

只要你的计算机使用ASCII字符集，就应该用整数65代替字符常量'A'。程序也应该完全以这种方式工作，但这样程序将会很难读。你要牢记，会有其他程序员要读你写的程序。除非这些程序员能记住ASCII表，否则看见程序中所写的整数65不会立刻联想到字母A。而字符常量'A'则可以更直接地传递出这个意义。

本书用表9-1给出了字符如何在机器内部表示的更具体的理解。一旦你心中有了这样一个概念，你应该忘记特定的字符代码，而集中于字符本身。

常见问题 避免在程序中用整数常量表示ASCII字符。所有的字符常量应该用单引号将此常量括起来表示，如'A'或'*'。

9.2.4 ASCII代码方案的重要特性

尽管考虑特定字符的代码不是很重要的，但以下两个ASCII表的结构特性是需要牢记的：

309～310

1) 表示数字0～9的字符的代码是连续的。尽管你不需要知道哪个代码对应数字字符'0'，但要知道数字'1'的代码是比'0'的代码大1的整数。同样，如果给'0'的代码加9，就是字符'9'的代码。

2) 字母按字母顺序分成两段：一段是大写字母（A～Z），另一段是小写字母（a～z）。然而，在每一段中，ASCII值是连续的，因此，为了得到它们的ASCII代码，你可以一个个字母

```
"Bother," said Pooh.
```

ASCII表中的许多特殊字符都没有明确的名字，在程序中使用它们的内部代码来表示。在此过程中，唯一的不便之处是这些特殊字符的数字代码是用八进制表示的，通常称为**八进制计数法**（octal notation）。在八进制计数法中，每一位数字是它右边数字的八倍。例如，字符常量'\177'表示ASCII值为八进制数177的字符（这个字符对应于表示Delete键的代码，在某些键盘上标记为Rubout）。在数值上，八进制值177对应于整数

$$1 \times 64+7 \times 8+7$$

结果为十进制数127。

ASCII代码系统中的许多特殊字符实际上很少使用。对大多数程序设计应用来说，只需要知道换行（'\n'）、tab（'\t'）和空字符（'\0'）就可以了，这些内容将在第14章讨论。

9.2.6 字符运算

在C语言中，字符值能像整数一样计算，不需要特别转换。结果是根据其内部ASCII代码定义的。例如，字符'A'在计算机内部用ASCII代码65表示，在运算时被当作整数65处理。 312

因为整数和字符能自由地相互转换，所以可以很容易地定义一个函数RandomLetter，返回一个随机选择的大写字母。有了第8章的random.h接口导出的RandomInteger函数，它的实现可写为

```
char RandomLetter(void)
{
    return (RandomInteger('A', 'Z'));
}
```

尽管对char类型的值应用任何算术运算都是合法的，但在它的值域内，不是所有运算都是有意义的。例如，在程序中将'A'乘以'B'是合法的。为了得到这个结果，计算机取它的内部代码，65和66，将它们乘起来，得到4290。问题在于这个整数作为字符毫无意义，事实上，它超出了ASCII字符的范围。当对字符进行运算时，仅有很少的算术运算是有用的。这些有意义的运算通常有：

- *给一个字符加上一个整数* 如果c是一个字符，n是一个整数，表达式c + n表示代码序列中c后面的第n个字符。例如，如果n在0～9之间，表达式'0' + n得到的是第n个数字的字符代码。因此，'0' + 5是'5'的字符代码。同样，如果n在1～26之间，那么'A' + n-1表示字母表中第n个字母的字符代码。
- *从一个字符中减去一个整数* 表达式c-n表示代码序列中c前面的第n个字符。例如，表达式'Z'-2的结果是'X'的字符代码。
- *从一个字符中减去另一个字符* 如果c1和c2都是字符，那么表达式c1-c2表示两个字符在代码序列中的距离。例如，如果你再回顾一下表9-1，并计算每个字符的ASCII值，你可以确定'a'-'A'是32。更重要的是，小写字母和与之对应的大写字母之间的距离是固定的，因此'z'-'Z'也是32。
- *比较两个字符* 用某个关系运算比较两个字符的值是常用的运算，经常用来确定字母的次序。例如，如果在ASCII表中，c1在c2前面，那么表达式c1<c2是TRUE。

toupper(ch) 如果ch是小写字母，返回它对应的大写字母；否则，返回ch本身。

虽然ctype.h接口已经提供了tolower和toupper两个函数，但如果你从头实现它们，则能更进一步了解它们的操作。同样，你可以忽略确切的ASCII代码，仅依赖于连续的假设。如果ch包含一个大写字母的字符代码，可以加上一个大写字母和小写字母的内码之间的差，将它转换为小写。然而，这里并没有把这个差作为一个确定的常量，如果把它表示成字符运算'a'-'A'，则程序的可读性更好。因此，你可以用如下代码实现这个tolower函数：

```
char tolower(char ch)
{
    if (ch >= 'A' && ch <= 'Z') {
        return (ch + ('a' - 'A'));
    } else {
        return (ch);
    }
}
```

函数toupper的实现方法类似。

尽管在ctype.h中定义的函数都很容易实现，但使用库函数而不是自己编写的函数是良好的程序设计习惯。这样做有三个主要的理由。

1) 因为库函数是标准的，其他程序员读你编写的程序比较容易。假设这些程序员都有C语言的经验，他们就知道ctype.h中的函数，并确切知道它们的含义。

2) 库函数比你自己写的函数更能保证正确性。因为ANSI C的库被成千上万个客户程序员所使用，实现者为保证函数的正确性作了大量的工作。如果你自己重写库函数，那么带来错误的机会更大。

3) 函数的库实现通常比你自己写的函数效率更高。例如，在ctype.h接口中，库中的机制比本节中给出的实现运行速度更快，通常是快三或四倍。这些高效的实现到底是如何工作的，这已超出本章的介绍范围，重要的是，你可以使用库提高程序的效率。

9.2.8 涉及字符的控制语句

因为char是标量类型，你可以把它用在可以出现整数的所有语句中。例如，如果ch声明为char类型，你可以用下列的for语句执行26次循环，按字母顺序对每个大写字母执行一次： 315

```
for (ch = "A"; ch <= "Z"; ch++)
```

同样，你也可以将字符用在switch语句的控制表达式中。例如，如果参数为元音的话，下面的谓词函数返回TRUE：

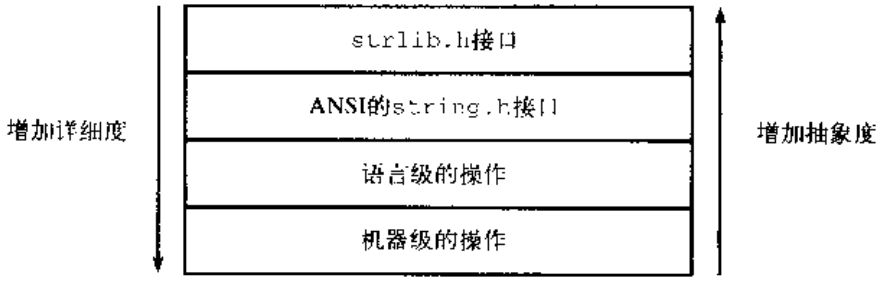
```
bool IsVowel(char ch)
{
    switch (tolower(ch)) {
      case 'a': case 'e': case 'i': case 'o': case 'u':
        return (TRUE);
      default:
        return (FALSE);
    }
}
```

注意，对大写和小写的形式，实现中都用tolower函数识别元音。

9.2.9 字符的输入输出

习惯上，字符的输入输出是用标准I/O库函数getchar和putchar完成的。本书将在第

建立在不同层次结构上的抽象称为**分层抽象**（layered abstraction）。用来表示字符串的分层抽象结构如下图所示： 317



完成输入输出的硬件设备自动在ASCII代码和屏幕或键盘上的符号之间进行转换。如本章前面所讨论，计算机对表示字符的整数代码应用算术运算可以处理每个字符。这些功能组成了可用于字符串的机器级的操作，形成层次结构中的最低一层。

在由硬件提供的基本功能的上面，程序设计语言通常也包括一些对字符串操作的支持。可用于字符串的内嵌操作形成了层次结构中的第二层。在许多语言中，这些功能是非常强大的，可以直接在语言级完成复杂的字符串操作。然而，ANSI C在语言本身几乎没有提供任何对字符串的支持。仅有的支持是提供定义字符串常量功能，这在第2章中的2.5.1节已经进行了介绍。所有其他的字符串操作都是由库提供的。甚至类型名string也不是语言的一部分，而是在genlib.h接口中作为扩展定义的。

在ANSI C中，程序员可用的大多数字符串操作是由string.h接口提供的。这个库提供了一组功能强大的字符串操作，但string.h接口中的概念抽象超出了你目前所学的知识，到学完第11章和第13章的内容后才能理解。然而，某些常用的操作用string.h接口难以完成。例如，当你用string.h中的函数时，不能很容易地从函数返回字符串值，或甚至直接为一个变量赋一个字符串值。因为使用string.h接口是ANSI C中字符串操作的标准方法，最终你必须学会如何使用它。本书将string.h接口放到第14章介绍，确保在你理解这个软件包的概念抽象前，你已经掌握了所有所需的背景知识。

为了让你有机会能用概念上较简单的模型处理字符串，本书引入另一个字符串库，它可以通过strlib.h接口访问。和genlib.h和stdio.h一样，strlib.h接口也是标准ANSI C库的扩展。因此，要把它包含到你的程序中，你必须要用引号：

```
#include "strlib.h"
```
318

strlib.h接口形成了抽象层次结构的最高层，使得字符串操作相对容易。

9.3.2 抽象类型的概念

strlib.h接口的主要优势在于它能使字符串作为*抽象类型*来处理。虽然这个定义将在第17章中细化（在第17章中抽象类型将作为一个中心话题），但目前我们可以把**抽象类型**（abstract type）想象为按照它的行为而不是按照它的表示定义的类型。

抽象类型的行为是由能在这种类型的对象上实施的操作来定义的。特定抽象类型的合法操作被称为它的**基本操作**（primitive operation），被定义为与该类型相关的接口中的函数。这些操作的细节和数据的基本表示都被隐藏在该接口的实现中。不管客户何时要操作某个抽象类型的值，客户必须用该接口提供的函数。

对字符串来讲，所需要完成的基本操作是什么呢？首先，你已经知道如何

下面的程序从用户那里读入一行文本，并报告它的长度：

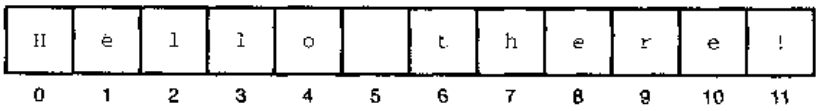
```
main()
{
    string str;

    printf("This program tests the StringLength function.\n");
    printf("Enter a string: ");
    str = GetLine();
    printf("The length is %d.\n", StringLength(str));
}
```

320

9.4.2 从一个字符串中选择字符

在C语言中，字符串中的位置是从0开始编号的。例如，在字符串"Hello there!"中，每个字符的编号如下图所示：



写在字符串的每个字符下面的正数称为它在字符串中的**下标**（index）。

为了使你能按给定的下标在字符串中选择特定的字符，strlib.h接口提供了一个函数IthChar，它有两个参数，一个字符串和一个表示下标的整数，该函数返回一个字符。例如，如果变量str包含字符串"Hello there!"，调用IthChar(str, 0)将返回字符'H'。同样，调用IthChar(str, 5)将返回' '，这是一个空格。记住，C语言中字符的编号是从0开始，而不是从1开始。这条规则很容易被忘记，会假设IthChar(str, 5)返回的是字符串中的第5个字符。IthChar(str, 5)返回的是下标位置为5的字符，是字符串中的第6个字符。

给出了库函数IthChar之后，你可以定义一个新的函数LastChar(str)，它返回str中的最后一个字符，如下所示：

```
char LastChar(string str)
{
    return(IthChar(str, StringLength(str) - 1));
}
```

例如，对字符串"Hello there!"调用LastChar将返回一个感叹号'!'。

9.4.3 连接

字符串的另一个有用的函数是Concat函数，它将两个字符串连接起来，当中不加任何字符。在程序设计中，这个操作被称为**连接**（concatenation）。例如，

```
Concat("Hello", "there")
```

的值是十个字符组成的字符串"Hellothere"。

如果要在两个单词间放一个空格，必须执行另一个连接。Concat函数每次只能取两个参数。为了连接三个或更多的字符串，必须多次调用Concat。每次调用连接两个字符串。例如，如果变量word1包含"Hello"，变量word2包含"there"，为了产生11个字符的字符串"Hello there"，必须嵌套调用Concat，如下列表达式所示：

321

```
Concat(Concat(word1, " "), word2)
```

以反序出现。下面的ReverseString的实现说明了这个策略：

```
string ReverseString(string str)
{
    string result;
    int i;

    result = "";
    for (i = 0; i < StringLength(str); i++) {
        result = Concat(CharToString(IthChar(str, i)), result);
    }
    return (result);
}
```

注意，每个字符在连接到变量result的开始处以前，必须被转换成字符串。

9.4.5 抽取字符串的一部分

连接将几个短的字符串形成长的字符串。你还经常要做相反的工作：将一个字符串分解为较短的字符串。一个字符串如果是某一较长的字符串的一部分，则被称为一个**子串**（substring）。strlib.h库提供了一个函数SubString(s, p1, p2)，它的作用是从s中抽取从位置p1到p2之间的字符，包括p1和p2。因此，函数调用

```
SubString("Hello there!", 1, 3)
```

返回字符串"ell"。如你所知，在C语言中字符编号是从0开始，因此，下标位置为1的字符是字符'e'。

作为SubString的一个应用实例，函数SecondHalf(s)返回一个由s中后半部分字符组成的子串，如果字符串长度是奇数，则包括中间的字符：

```
string SecondHalf(string str)
{
    int len;

    len = StringLength(str);
    return (SubString(str, len / 2, len - 1));
}
```

SubString函数对一些特殊情况处理如下：

323

1) 如果p1是负数，它会被设为0，使它指向字符串的第一个字符。

2) 如果p2大于StringLength(s)-1，它会被设置为StringLength(s)-1，使它指向最后一个字符。

3) 如果p1大于p2，则SubString返回一个空串。

虽然在你对字符串没有更多的经验前可能不太清楚选择这种设计的理由，但当你调用SubString时，这些规则使你不需要检查一些特殊情况，使字符串编程更容易。

9.4.6 比较两个字符串

从第5章开始，你已经能用StringEqual函数比较两个字符串是否完全相同。在许多情况下，你还会发现确定两个字符串按字母顺序的先后位置也是非常有用的。strlib.h接口提供了一个函数StringCompare实现这个功能。StringCompare有两个字符串参数s1和s2，它返回一个整数，该整数的符号指出了两个字符串的关系：

- 如果按字母顺序，s1出现在s2的前面，则StringCompare返回一个负整数。

strlib.h接口提供了两个函数FindChar和FindString来实现这个功能。FingChar的原型是

```
int FindChar(char ch, string text, int start);
```

该函数搜索字符串text，从start指定的下标位置开始，寻找字符ch的第一次出现。如果发现了该字符，FindChar返回这个字符的下标位置。如果在text结束前没有发现这个字符，FindChar返回值-1。下面的例子说明了这个操作（记住，下标号从0开始）：

```
FindChar('l', "Hello there", 0)   (返回2)
FindChar('l', "Hello there", 3)   (返回3)
FindChar('l', "Hello there", 4)   (返回-1)
```

325

和字符串比较一样，搜索字符串的函数对大小写是区别对待的。因此，调用

```
FindChar('h', "Hello there", 0)
```

返回7，因为小写字母h的第一次出现在下标位置7。在下标位置0的大写字母H被忽略了。

你可以用FindChar实现一个产生**首字母缩写**（acronym）的函数，它按次序取一系列单词的首字母形成一个新的单词。例如，单词scuba是由self contained underwater breathing apparatus的首字母形成的首字母缩写词。函数Acronym取一个由若干个独立的单词组成的字符串，返回它们的首字母缩写词。因此，调用函数

```
Acronym("self contained underwater breathing apparatus")
```

将返回"scuba"。

只要单词用单个空格分开，并且没有其他字符出现，Acronym的实现就非常简单，它取最前面的字母，然后进入循环，搜索空格。一旦发现了空格，就将下一个字符连到保存结果的字符串变量的最后。当字符串中找不到空格时，首字母缩写词就完成了。这个策略可以翻译成下列C语言的实现：

```
string Acronym(string str)
{
    string acronym;
    int pos;

    acronym = CharToString(IthChar(str, 0));
    pos = 0;
    while (TRUE) {
        pos = FindChar(' ', str, pos + 1);
        if (pos == -1) break;
        acronym = Concat(acronym,
                         CharToString(IthChar(str, pos + 1)));
    }
    return (acronym);
}
```

函数FindString(str, text, start)与FindChar相似，除了第一个参数是字符串以外。函数搜索字符串text，从位置start开始寻找字符串str。如果找到匹配的字符串，FindString返回匹配开始处的下标位置。例如，

```
FindString("there", "Hello there", 0)
```

将返回值6。如果没有找到匹配的字符串，和FindChar一样，FingString返回-1。

例如，函数ReplaceFirst(str, pattern, replacement)搜索字符串str，将字

果n是负数，前面再加一个负号。例如，调用IntegerToString(123)将返回字符串"123"，调用IntegerToString(-4)将返回字符串"-4"。RealToString(d)函数将浮点数d转换为字符串，该字符串的格式类似于由printf用格式码%G的显示结果，有时会产生一个以科学记数法表示的数字。例如，调用RealToString(3.14)返回“3.14”，而调用RealToString(0.00000000015)将返回"1.5E-10"。

```
/*
 * File: repfirst.c
 * ----------------
 * This file implements and tests the function ReplaceFirst.
 */

#include <stdio.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/* Function prototypes */

string ReplaceFirst(string str, string pattern, string replacement);

/* Main program */

main()
{
    string str, pattern, replacement;

    printf("This program edits a string by replacing the first\n");
    printf("instance of a pattern substring by a new string.\n");
    printf("Enter the string to be edited:\n");
    str = GetLine();
    printf("Enter the pattern string: ");
    pattern = GetLine();
    printf("Enter the replacement string: ");
    replacement = GetLine();
    str = ReplaceFirst(str, pattern, replacement);
    printf("%s\n", str);
}

/*
 * Function: ReplaceFirst
 * Usage: newstr = ReplaceFirst(str, pattern, replacement);
 * --------------------------------------------------------
 * This function searches through the string str and replaces the
 * first instance of the pattern with the specified replacement.
 * If the pattern string does not appear, str is returned unchanged.
 */

string ReplaceFirst(string str, string pattern, string replacement)
{
    string head, tail;
    int pos;
```

图9-1 repfirst.c

```
        total += StringToInteger(line);
    }
    printf("The total is %d\n", total);
}
```

由于GetInteger不能读一个空行作为数据，因此在第3章的addlist.c中没有将空行作为输入结束标记的一种选择。如果程序用GetLine读入输入值，空行将作为一个空串出现。如果从用户读入的行不是空行，那么该行中的字符将转换成整数，并加到运行结果中。

9.4.10 效率和strlib.h库

本章中所有函数的实现都是以清晰，而不是以效率作为目标。当你学完第14章中各个字符串库的内部细节时，你将发现本章介绍的许多实现的效率都是相当低的，低得无法适应一些重大的应用。然而，它们是简明的、可行的、容易理解的。当用这种形式的函数工作时，你可以从概念上了解字符串工作的过程。当你阅读strlib.h接口的实现并研究它如何工作时，这些知识是非常有帮助的。 330

小结

在本章中，你已经开始理解如何使用*文本数据*。在C语言中，文本数据最常见的形式是*字符串*，它是单个字符的有序集合。单个字符在C语言中用数据类型char表示，它是称为*标量类型*的一类数据的一部分。标量类型也包括*枚举类型*，它也是在本章中介绍的。使用字符是处理字符串的基础，因为字符是形成字符串的“原子”。

在本章中，你学习了如何将字符串作为*抽象类型*来操作。抽象类型的内部表示被隐藏起来。客户可以通过调用在某个接口中定义的函数来使用该类型的值。在字符串的情况中，提供字符串操作的接口是strlib.h，它形成了*分层抽象*的最高层次。通过学习如何将字符串作为抽象数据类型来操作，你可以逐步发现如何以令人兴奋的和复杂的方法使用它们，尽管它的结构是不可见的。

然而，因为字符串是现代程序设计的基础，有必要从不同的角度去了解它。所以在第11章、第13章和第14章还将学到更多的有关字符串以及它们的表示的知识。

本章的重点内容包括：

- 值不是数字的类型在计算机内部通常通过对该类型的值域中的元素编码，然后用这些编号作为原先值的代码来表示。通过对元素进行计数而定义的类型称为*枚举类型*。
- C语言用关键词typedef和enum定义新的枚举类型。
- 枚举类型是*标量类型*的子类，标量类型的行为与整数的行为完全一样。在C语言中，标量类型和整数类型之间不需要显式的转换。
- 按照预先定义的*字符代码*，字符在内部被表示为整数。虽然某些计算机使用不同的字符编码系统，但大多数现代的计算机都是用表9-1所示的ASCII编码系统。
- 用来表示单个字符的数据类型char是一个标量类型。因此，字符值可以用标准的算术运算来处理。
- ctype.h接口包含一些为单个字符分类，及改变单个字符大小写的函数。
- strlib.h接口使字符串能作为*抽象类型*工作。
- strlib.h接口定义了一些处理字符串的函数。这些函数归纳在表9-3中。 331

符，尽管它们的类型是int。执行实现中的这些步骤，验证在表示字符的所有值上完成的操作确实是在上一个问题中列出的这些操作。

12. 调用isdigit(5)的结果是什么？调用isdigit('5')的结果是什么？

13. 调用toupper('5')的结果是什么？

14. 判断题：在switch语句中，使用字符常量作为case表达式是合法的。

15. 什么是分层抽象？

16. 判断题：抽象类型是按其行为，而不是按其表示定义的。

17. 调用下列函数的结果是什么？

a. StringLength("ABCDE")

b. StringLength("")

c. StringLength("\a")

d. IthChar("ABC", 2)

e. Concat("12", ".00")

f. CharToString('2')

g. SubString("ABCDE", 0, 3)

h. SubString("ABCDE", 4, 1)

i. SubString("ABCDE", 3, 9)

j. SubString("ABCDE", 3, 3)

18. 如果你要在一个已有的字符串后添加一个字符，应该用strlib.h中的什么函数？ 333

19. 当比较字符串时，必须牢记哪些重要的注意事项？

20. 调用下列函数的结果是什么？（对调用StringCompare，只需指出结果的符号。）

a. StringEqual("ABCDE", "abcde")

b. StringCompare("ABCDE", "ABCDE")

c. StringCompare("ABCDE", "ABC")

d. StringCompare("ABCDE", "abcde")

e. FindChar('a', "Abracadabra", 0)

f. FindString('ra', "Abracadabra", 3)

g. FindString("is", "This is a test.", 0)

h. FindString("This is a test", "test", 0)

21. 调用下列函数的结果是什么？

a. ConvertToLowerCase("Catch-22")

b. StringToInteger(SubString("Catch-22", 5, 7))

c. RealToString(3.140)

d. Concat(IntegerToString(4/3), "pi")

程序设计练习

1. 定义一个函数LeftFrom(dir)，它有一个类型为directionT的参数，表示指南针上的点，返回一个左转90度后指南针上的点。为你的函数写一个测试程序，显示对指南针上的每个点调用LeftFrom后的结果。如下所示：

写一个主程序测试ScrabbleScore函数。这个测试程序的一次可能的运行示例如下所示：

```
This program tests the ScrabbleScore function.
Enter words, ending with a blank line.
Word: XI↵
The basic score for 'XI' is 9.
Word: HORN↵
The basic score for 'HORN' is 7.
Word: SCRABBLE↵
The basic score for 'SCRABBLE' is 14.
Word: QUIzZICAL↵
The basic score for 'QUIzZICAL' is 28.
Word:
```

6. 实现一个函数Capitalize(str)，它返回一个首字符是大写的（如果是字母），其他所有字母都转换成小写的字符串。其他字符不受影响。例如，Capitalize("BOOLEAN")和Capitalize("boolean")应该都返回字符串"Boolean"。

7. 实现一个谓词函数EqualIgnoringCase(s1, s2)，在不考虑大小写的情况下，如果字符串s1和s2是相同的，则返回TRUE。例如，EqualIgnoringCase("CAT", "cat")应该返回TRUE。

8. 使人们难以读出信息的最简单的编码方法之一是**字母替换加密**（letter-substitution cipher），原始消息中的每个字母都由不同的字母来替换。字母替换加密中最简单的一种是**循环加密**（cyclic cipher）。其中每个字母都用字母表中在该字母之后的、与其相隔固定距离的另一字母替代。单词循环指的是如果操作到字母表开始的字符时，则通过Z再次循环到字母表开始，重新从A开始。

作为一个实例，下面的运行示例显示了字母表中的每个字母向前移4个字母。A变成了E，B变成了F，Z变成了D（因为它循环到字母表的开始位置），等等。

```
This program encodes a message using a cyclic cipher.
Enter the numeric key: 4↵
Enter a message: ABCDEFGHIJKLMNOPQRSTUVWXYZ↵
Encoded message: EFGHIJKLMNOPQRSTUVWXYZABCD
```

336

为了解决这个问题，应该先定义一个函数EncodeString，它的原型为

```
string EncodeString(string str, int key)
```

该函数返回一个新的字符串，它是由将str中的每个字母后移key个字母而形成的，必要时，还要返回到字母表的开始位置。实现EncodeString后，编写一个测试程序，重复下面的运行示例：

```
This program encodes a message using a cyclic cipher.
Enter the numeric key: 13↵
Enter a message: This is a secret message.↵
Encoded message: Guvf vf n frperg zrffntr.
```

注意，编码操作仅用于字母；其他字符在输出时保持不变。此外，字母的大小写也不受影响；小写字母在编码后还是小写字母，大写字母在编码后也还是大写字母。

写一个程序，使得负的key值表示字母向字母表的开始位置移动，而不是向字母表的结

第10章 模块化开发

> It is particularly important, it seems to me, in an era of ever increasing departmentalization and specialization, to make the attempt occasionally to see wholes and to understand what lies behind the exterior manifestations.
>
> — Jessamyn West, *The Quaker Reader*, 1962

学习目标

- 认识把单个程序分为多个单独模块的重要性。
- 理解在一个模块里需要在多个函数调用之间保存状态信息。
- 能够使用全局变量来表示在跨函数调用中所需维护的状态信息。
- 认识到过度使用全局变量的危险性。
- 能够使用static关键字来保证一个模块中的函数和全局变量的私有性。

339

目前为止，本书给出的大部分程序都是一些短小的程序，用来说明C语言的一些特殊方面。关注单个语句格式或者其他一些语言细节对于初学编程的人来说是有意义的。这样读者能够单独考虑每个概念，并且了解它是如何工作的，而不必涉及大程序中的内在的复杂性。但是，真正具有挑战性的则是如何掌握这种复杂性。为了达到这个目的，就必须写一些大的程序，在这样的程序中，将一些单独的概念和工具结合起来。

管理大程序的最重要的技术是第5章所学的逐步精化策略，即当遇到一个需要复杂解决方案的较大的问题时，把问题分解成更为容易解决的几个小部分。

把一个大的问题分解成小片段的策略的方法在程序设计过程的很多阶段都会用到。第5章中的calender.c实例在只有一个源文件的情况下阐述了此项技术。然而，当程序变得更长的时候，要在一个源文件中处理如此众多的函数会变得困难。就像要一下子理解一个50行的函数也是比较困难的一样。在这两种情况下，引入一些额外的结构是很有用的。比如碰到一个50行的函数时，最好的方法是把它分成几个相互调用的小函数来完成任务。当碰到一个包含50个函数的程序时，最好的办法就是把程序再分成几个更小的源文件。每个源文件都包含一组相关的函数。由整个程序的一部分组成的较小的源文件称为**模块**（module）。每个独立的模块比整个程序要简单。而且，如果在设计的时候非常仔细的话，可以把同一模块作为许多不同的应用程序一部分。

program.c

```
main()
{
   ProcA();
   ProcB();
}

void ProcA(void)
{
   . . .
}

void ProcB(void)
{
   . . .
}
```

图10-1 作为单个模块的程序

340

当把一个程序分成模块的时候，选择合适的分解方法来减少模块之间相互依赖的程度是很重要的。包含main函数的模块叫**主模块**（main module），在分解层次中处于最高层。每个其他模块代表一个独立的抽象，其操作在一个接口中定义。

2) 如果单词以元音开头，则加后缀way。

例如，假设单词是scram。因为是以辅音开头，所以把它分为两部分。第一部分由第一个元音之前的所有字符组成，另一部分包括那个元音以及之后的字母，（如下图所示）。



然后交换这两部分的位置，并且在末尾加上ay（如下图所示）。



这样就创建了Pig Latin单词amscray。对于以元音开头的单词，如apple，只要在末尾加上way，从而得到appleway。

10.1.1 应用自顶向下的设计

既然问题是把整行英文翻译成Pig Latin，那么程序应该能够产成如下运行示例：

```
Enter a line: this is pig latin.↵
isthay isway igpay atinlay.
```

342

在开始设计Pig Latin问题的解决方案时，可以先不立即决定是否需要把程序分割成的模块。通常，仅仅在比较了不同的解决方法以后，才可以决定一个程序是否需要一个特定模块分解。通常，最好的方法是先应用自顶向下的设计方法，如5.6节所描述的那样。

采用自顶向下的设计方法，从主程序的层次着手，然后依次写下一系列函数，每个函数解决整个问题中的一个部分。在初始阶段，可以用一系列有待具体实现的高层步骤来定义main函数。虽然可以将这些步骤直接编码为函数调用，但是通常先描述出其功能更为容易。例如，当你在纸张上设计程序时，你可以写出如下的Pig Latin程序的初稿：

```
main()
{
    Read in a line of text from the user.
    Translate the line of text into Pig Latin.
}
```

在这个例子里，函数头和花括号是C语言标准语法的一部分，而语句本身则写成英文语句，用来描述程序要做什么事情。由英文和C语言混合构成的程序称为**伪代码**（pseudocode）。

10.1.2 使用伪代码

虽然伪代码对编译器来说没有意义，但是对于程序员非常有用，因为它可以记录逐步精化的过程。在用一系列英文步骤写出整个程序之后，可以再回到伪代码语句，用实际的C语言代码来实现它们。例如，在这个例子中，可以很容易地把第一句伪代码翻成C语句，因为这和第2章以来我们一直使用的读取一个字符串的习语相吻合。在填写完从用户处读入一行文本的操作之后，程序的伪代码版本如下所示：

```
main()
{
    string line;

    printf("Enter a line:");
```

你学完第11章的时候，你会有用来处理单词列表的工具，但到目前为止，你必须使用现在所具备的工具来完成任务。如果仔细考虑这个问题，会发现不需要立即记录所有的单词。一旦找到一个单词，就可以马上翻译并且正确显示。一旦显示出这个单词，就可以丢开它继续处理下一个单词。根据这个观察结果，可以提出第二种策略：

```
Void TranslateLine(string line)
{
    while(there are any words left on the line){
        Get the next word.
        Translate that word into Pig Latin.
        Display the translated word.
    }
    Display a newline character to complete the output line.
}
```

虽然这个策略的伪代码里还有很多细节问题没有考虑，但是总体思想看起来是有意义的，并且可以避免记录整个单词列表的问题。

10.1.4 考虑空格和标点符号的问题

在TranslateLine的伪代码版本所用的策略中有个小问题。例如，在程序运行时，假设用户输入了以下数据行：

```
Enter a line: this is pig latin.↵
```

如果把输入看作四个单词this、is、pig和latin，如果所有的英文表示的步骤都如预期的那样正常运行，那么程序输出将会是：

```
Enter a line: this is pig latin.↵
isthayiswayigpayatinlay
```

345

这个输出并不是我们真正想要的。因为没有空格和标点符号，所以所有的单词都挤在一起了。这个伪代码版本没有把空格和标点符号考虑进去。而原来对问题的英文陈述也没有考虑这个问题。这个问题没有被完整地定义过。

程序设计中的一个现实问题是问题的英语描述通常是不完整的。作为一个程序员，常常会被一些所忽略的或者认为太明显而没有提及的细节所难倒。在某些情况下，被省略的部分是非常重要的，需要跟给你布置任务的人进一步讨论。然而，在很多时候，你必须自己选择一种合理的策略。

但是，判断什么是合理的策略也有一些技巧。在这个例子中，需要决定的是在输出的每个单词之间打印空格，而忽略其他标点符号。这个策略很简单，而且在本例中很合理。但另一方面它又可能不是最好的。标点符号可以提高输出的可读性。因为标点符号和空格都能表达出某种意义，所以最好让它们在输出中的位置和在输入中的位置相同。因此，需要的输出为：

```
Enter a line: this is pig latin.↵
isthay isway igpay atinlay.
```

```
        Get the next token.
        if (the token is a regular English word) {
            Replace the token by its Pig Latin translation.
        }
        Display the token.
    }
    Display a newline character to complete the output line.
}
```

这个策略仍然是相当简单的。而且，获取一个记号并测试这些记号是否留在该行中的操作很可能在不同的应用程序中都是有用的。

把一行分成一些独立的记号的想法在计算机科学中非常常见。例如，当C编译器将一段程序翻译成机器代码的时候，此过程的第一步就是将输入文件分割成C语言所用的记号，即变量名、数字、运算符等等。将输入分解成记号的过程称为**词法分析**（lexical analysis），或者更通俗地说，把它称为**记号扫描**（token scanning）。

10.1.6 设计记号扫描器

要使用新的策略来完成TranslateLine的实现，必须首先设计能够分割输入行的记号扫描器。而且，要记住它很可能是个通用的工具，除了将字符串转换成Pig Latin外，记号扫描器还能用于许多其他问题。因此，将记号扫描器设计成一个独立的模块就十分有意义。起初，设计过程需要考虑如何组织扫描器模块，特别是它该包含什么函数。

在前面一节给出的最新的TranslateLine伪代码版本中，扫描器模块有两个不同的作用。第一，扫描器必须提供一个函数返回当前行中的下一个记号。第二，当扫描完最后一个记号时要让客户知道。因为你正在设计这个接口，所以你要选择这些函数的名字。负责返回下个记号的函数可以命名为GetNextToken。要报告所有记号都已经被扫描，一种选择是定义一个叫AtEndOfLine的谓词函数，它在读到最后一个记号时返回TRUE。

但现在你面临一个有趣的问题：这些函数的参数是什么呢？初一看，调用程序必须将输入行传递给GetNextToken，因为记号是来自于那一行的。另一方面，调用GetNextToken(line)的想法在概念上有矛盾之处。如果GetNextToken像通常的函数那样运作，那么GetNextToken(line)每次都返回同样的结果，因为line的值没有改变。

为了解释这个问题，我们假设把字符串"Hello there"分成三个记号：单词"Hello"，跟在它后面的空格，以及单词"there"。另外，假设已经用赋值语句

```
line = "Hello there";
```

把字符串"Hello there"存储在变量line中。如果GetNextToken将line作为参数，很容易就能想到调用GetNextToken(line)将得到第一个单词。问题是如何得到下一个记号。如果你再次调用GetNextToken(line)，变量line仍然包含着整个字符串"Hello there"。既然这个函数调用跟第一次出现的形式完全相同，那么它将返回同一个值。

为了改正这个问题，必须将扫描器模块设计为能够追踪划分行的进度。在GetNextToken从行返回一个记号后，必须记住已经扫描过的那些记号，以便在下次调用时返回一个不同的结果。一个模块中的多次函数调用之间的信息称为**内部状态**（internal state）。

当模块维护内部状态时，模块的接口通常导出一个用来初始化此状态信息的函数。例如，在扫描器模块中，提供一个函数InitScanner(line)是有用的，它使得客户能够告诉扫描器在字符串line的开头开始返回记号。为了得到这些记号，客户只要调用GetNextToken

```
bool IsLegalWord(string token)
{
    int i;

    for (i = 0; i < StringLength(token); i++) {
        if (!isalpha(IthChar(token, i))) return (FALSE);
    }
    return (TRUE);
}
```

翻译一个单词的过程稍微有些难。在伪代码中，TranslateWord的结构对应了Pig Latin的规则。

```
string TranslateWord(string word)
{
    Find the position of the first vowel.
    if (the vowel appears at the beginning of the word) {
        Return the word concatenated with "way".
    } else {
        Extract the initial substring up to the vowel and call it the "head."
        Extract the substring from that position onward and call it the "tail."
        Return the tail, concatenated with the head, concatenated with "ay".
    }
}
```

350

在伪代码中，第一条英文语句是最难实现的，而且是唯一对于定义一个新函数有意义的语句。函数FindFirstVowel(word)返回word的第一个元音的下标位置。这样，在TranslateWord实现中的第一条语句可以写成如下形式，其中vp是个用来记录元音位置的整型变量：

```
vp-FindFirstVowel(word);
```

如果你在设计中非常仔细，就会注意到函数FindFirstVowel(word)还没有完整的定义。非形式化的描述没有包含所有可能的情况。如果word中没有元音怎么办？在这种情况下，FindFirstVowel仍然要返回一些东西。库函数FindChar和FindString使用－1作为一个特殊的标志，用来指出函数查询的值在字符串中不存在。这里也可以采用相同的策略。

当FindFirstVowel返回－1时，TranslateWord必须有合理的响应，因为转换一个不包含元音的单词的结果还没有明确定义过，因此需要确定TranslateWord到底该做什么。最简单的办法是让TranslateWord返回原来的单词。结合这个设计决策意味着TranslateWord的代码将以如下行开始：

```
vp = FindFirstVowel(word);
if (vp == -1) {
    return (word);
} else . . .
```

TranslateWord接下来的步骤只是从strlib.h接口中调用合适的函数，这样使得编码过程很容易完成，如下所示：

```
string TranslateWord(string word)
{
    int vp;
    string head, tail;

    vp = FindFirstVowel(word);
    if (vp == -1) {
```

```
 *
 * When the last token has been read, the predicate function
 * AtEndOfLine returns TRUE, so that the loop structure
 *
 *        while (!AtEndOfLine()) {
 *            token = GetNextToken();
 *            . . . process the token . . .
 *        }
 *
 * serves as an idiom for processing each token on the line.
 *
 * Further details for each function are given in the
 * individual descriptions below.
 */

#ifndef _scanner_h
#define _scanner_h

#include "genlib.h"
/*
 * Function: InitScanner
 * Usage: InitScanner(line);
 * -------------------------
 * This function initializes the scanner and sets it up so that
 * it reads tokens from line.  After InitScanner has been called,
 * the first call to GetNextToken will return the first token
 * on the line, the next call will return the second token,
 * and so on.
 */

void InitScanner(string line);

/*
 * Function: GetNextToken
 * Usage: word = GetNextToken();
 * -----------------------------
 * This function returns the next token on the line.
 */

string GetNextToken(void);

/*
 * Function: AtEndOfLine
 * Usage: if (AtEndOfLine()) . . .
 * -------------------------------
 * This function returns TRUE when the scanner has reached
 * the end of the line.
 */

bool AtEndOfLine(void);

#endif
```

图 10-3 (续)

出现后的其余部分。

和局部变量不同，全局变量以某一种方式保持在内存中，在这种方式下它的值不受函数调用的影响。仍以索引卡片来比喻，全局变量被保存在一个始终有效的独立的卡片上，好像被粘在桌面上，永远都不会被包含局部变量的卡片覆盖一样。模块中每个函数都可以看到全局变量卡片的那些变量。而且，当函数返回时，这些变量值不会丢失。全局变量保持相同的值直到给它赋予新值为止。 355

10.2.2 使用全局变量的危险性

许多程序员觉得全局变量的想法非常有吸引力。毕竟，全局变量可以被程序的所有部分看到。不需要将它们作为参数来传给某个函数。但是，程序员很快会发现使用全局变量会使得代码很难读懂。

奇怪的是，新手觉得优越的特性往往是有经验的程序员非常认真对待的问题。新手喜欢全局变量可以被源文件中任何函数操作的特点。而对有经验的程序员来说，这是一个危险的信号。例如，你在查找一个由于变量被错误赋值而导致的程序错误。如果是个全局变量，由于模块中的每个函数都能操作那个变量，那么问题可能存在于源文件的任何位置。而局部变量的错误位置则更容易确定。如果局部变量值错误，程序员只需要查找应用该变量的函数即可。

为了避免这样的问题，在结构良好的程序中，一般很少使用全局变量。它们的主要优点在于使得像扫描器模块这样的模块可以维护内部状态。既然全局变量在函数调用之间可以维护它们的值，那么对于这样的用途非常理想。于是，在扫描器模块中，你可以用全局变量来追踪传给InitScanner的行和此行的当前位置。

常见错误 *在编写程序时，要避免使用全局变量，因为这样会使程序难以理解、难以调试。只有当模块必须维护函数调用之间的内部状态时，才必须使用全局变量。*

10.2.3 保持变量的模块私有化

全局变量在一个模块的任何地方都可见只是使用它们的问题之一。除非明确地声明，否则C编译器假定其他模块也可以看到全局变量。这样，当声明了一个全局变量后，可能改变其值的函数不只是局限在一个模块中。该变量可能被整个程序的任何模块引用。

在一个结构良好的程序中，独立的模块之间通过在模块间传递参数的函数调用来交换数据。在大多情况中，尤其是刚开始养成设计程序习惯的时候，最好确保每个全局变量不会被一个以上的模块引用。为了避免两个模块引用同一个全局变量的可能性，应该在声明前用关键字static来彻底避免这种危险。如：

```
static int cpos;
```

这个声明定义cpos为一个全局整型变量，在所定义的模块里的任何地方都可见。然而，cpos对于别的模块是无效的，因此它是当前模块私有的。在本书中，所有的全局变量都用static来声明。

在C语言中，单词*static*用来指示变量如何存储，这会在第11章中讨论。而在大多数实际应用中，最好认为*static*就是*prviate*的同义词，这样能更贴切地描述它的用途。用关键词static 356
声明变量可以使它们对于应用它们的函数来说是私有的。

10.2.4 初始化全局变量

在scanner.h接口中，通过调用InitScanner函数来初始化全局变量。使用一个初始

指出客户想要的扫描器的常规行为，即返回大小写混合的记号，就和输入行中出现的这些字符一样。

然而，这种方法有两个严重缺点。首先，程序阅读者很难知道在调用InitScanner时TRUE和FALSE参数的含义。为了理解这些参数的用途，任何客户都不得不阅读接口注释。第二，新的设计改变了已有的接口而破坏了稳定性。如果扫描接口已经有客户，那么这些客户不得不修改程序。

避免以上两种问题的更好的办法是扩展扫描器接口而非改变它。所有老的函数都像以前那样工作。为了提供返回大写记号的选项，可以增加一个新的函数ReturnUppercaseTokens，它带有一个布尔值，用它来设置uppercaseFlag。因此，调用

```
ReturnUppercaseTokens(TURE);
```

客户可以选择返回大写记号的新的行为模式。

要注意的是，如果ReturnUppercaseTokens从来未被调用，那么模块必须按照以前那样继续工作。毕竟，使用老的scanner.h接口的现有程序中没有一个程序会调用ReturnUppercaseTokens。甚至写这些客户程序时，这个函数还不存在。因此，即便没有明确的调用来设置它的值，变量uppercaseFlag必须用FALSE作为其初值。为了保证它有恰当的值，需要使用静态初始化： 358

```
static bool uppercaseFlag = FALSE;
```

本章的练习8给你一个机会来完成ReturnUppercaseTokens扩展的实现。

除非客户专门采取行动去改变否则一直在程序里使用的值称为**默认值**（default value）。在一个典型的模块中，指定客户可以设置的选项的全局变量通常被静态初始化为其默认值。客户通过调用接口提供的函数来改变这些值。

10.2.5 私有函数

关键字static除了用于全局变量外，还可以用来指示某函数是某个特定模块的私有函数。定义接口时候，接口导出的函数不是私有的。接口的要点就是让这些函数可以在其他模块中调用。在很多情况下，接口还包括一些只能在当前模块中调用的函数。要指出某一函数是否被限制在一个特定的模块中，可以把关键字static放在函数原型和其实现的前面。这样使得客户无法调用这些函数，从而使接口与用户间的抽象边界更加坚固稳定。

声明函数为static在由几个程序员参与开发的大型程序环境中也有好处。如果函数或者全局变量没有声明为static，组成整个程序的模块集合中的其他模块就不能使用这些名字。在不同函数中，避免名字相互干扰就需要独立模块的开发者之间必须相互沟通，确保不使用相同的名字，或者使用static关键字来保证他们使用的名字对于自己的模块的私有化。

如下的规则对于模块化开发来说是极好的指导：

> **静态声明规则** 除了main函数和接口明确导出的函数之外，所有的函数必须被声明为static。

从现在开始，本书所有的程序都遵循静态声明规则。

10.3 实现扫描器抽象

一旦理解模块的内部状态可以用全局变量保存，那么实现扫描器抽象的实际流程就变得

```
/*
 * File: scanner.c
 * ---------------
 * This file implements the scanner.h interface.
 */

#include <stdio.h>
#include <ctype.h>

#include "genlib.h"
#include "strlib.h"
#include "scanner.h"

/*
 * Private variables
 * -----------------
 * buffer  -- Private copy of the string passed to InitScanner
 * buflen  -- Length of the buffer, saved for efficiency
 * cpos    -- Current character position in the buffer
 */

static string buffer;
static int buflen;
static int cpos;

/*
 * Function: InitScanner
 * ---------------------
 * All this function has to do is initialize the private
 * variables used in the package.
 */

void InitScanner(string line)
{
    buffer = line;
    buflen = StringLength(buffer);
    cpos = 0;
}

/*
 * Function: GetNextToken
 * ----------------------
 * The implementation of GetNextToken follows its behavioral
 * description as given in the interface: if the next character
 * is alphanumeric (i.e., a letter or digit), the function
 * searches to find an unbroken string of such characters and
 * returns the entire string.  If the current character is not
 * a letter or digit, a one-character string containing that
 * character is returned.
 */

string GetNextToken(void)
{
    char ch;
    int start;

    if (cpos >= buflen) Error("No more tokens");
```

361

图10-4 scanner.c

```
/* Main program */

main()
{
    string line;

    printf("Enter a line: ");
    line = GetLine();
    TranslateLine(line);
}

/*
 * Function: TranslateLine
 * Usage: TranslateLine(line);
 * ---------------------------
 * This function takes a line of text and translates
 * the words in the line to Pig Latin, displaying the
 * translation as it goes.
 */

static void TranslateLine(string line)
{
    string token;

    InitScanner(line);
    while (!AtEndOfLine()) {
        token = GetNextToken();
        if (IsLegalWord(token)) token = TranslateWord(token);
        printf("%s", token);
    }
    printf("\n");
}

/*
 * Function: TranslateWord
 * Usage: word = TranslateWord(word)
 * ---------------------------------
 * This function translates a word from English to Pig Latin
 * and returns the translated word.
 */

static string TranslateWord(string word)
{
    int vp;
    string head, tail;

    vp = FindFirstVowel(word);
    if (vp == -1) {
        return (word);
    } else if (vp == 0) {
        return (Concat(word, "way"));
    } else {
        head = SubString(word, 0, vp - 1);
        tail = SubString(word, vp, StringLength(word) - 1);
        return (Concat(tail, Concat(head, "ay")));
```

363

图 10-5 (续)

小结

在本章中，介绍了一些新的技术，利用这些技术可以容易地将一个应用分成多个独立的模块来构成整个程序。特别是如何在模块中表示内部状态，这使得模块中的函数可以在函数调用之间保持某些效用。

本章的重点内容包括：

- 大型程序可以通过分割成独立的模块来加以简化。每个模块写成一个单独的源文件。包含main函数的模块是主模块。
- 当试图通过自顶向下的设计来解决问题的时候，对程序进行有效的模块分解是十分必要的。
- 设计函数时，最好把各个步骤用英文和C语言写出来，这种代码被称为伪代码。
- 模块常常需要维护独立函数调用之间保持的内部状态。
- 内部状态必须用全局变量来保存。一个模块中的所有函数都可以访问全局变量。和局部变量不同，全局变量在函数返回时不会消失。
- 使用全局变量会使程序变得难以阅读和调试。为了避免这些问题，应该仅在需要维护模块内部状态时候使用全局变量。
- 所有用来记录内部状态的变量都应该声明为在其所出现的模块中是私有的。可以通过在变量声明的开始处使用static关键字来实现。
- 除非函数是由接口导出的。否则应该使用static关键字来声明该函数是当前模块中私有的。

复习题

1. 判断题：每个模块对应于一个独立的源文件。
2. 怎么判断几个模块中哪个是主模块？
3. 判断题：在开始实施自顶向下的设计之前，总是该选择对程序进行模块分解。
4. 术语伪代码是什么意思？
5. 当发现问题没有被完整定义的时候该怎么做？
6. 什么是记号？ 366
7. 局部变量和全局变量之间的主要区别是什么？
8. 阅读程序的时候，如何判断变量是局部的还是全局的？
9. 局部变量和全局变量的作用域分别是什么？
10. 为什么过度使用全局变量是有危险的？
11. 关键字static的功能是什么？
12. 静态初始化和动态初始化的区别是什么？什么时候使用静态初始化最有用？

程序设计练习

1. 在几个世纪以前，缩写的形成和现在不同。一种古老的缩写风格不是由单词的首字符组成，而是由前面的辅音（或者如果单词以元音开头则取第一个字母），再跟上单词最后的字母组成。这种风格在美国很多州名的缩写（不是两个字母的邮政缩写）中很常见，例如：

Connecticut	Ct.	Maine	Me.
Florida	Fla.	Pennsylvania	Pa.

6. 写一个函数LongestWord(line)，返回line中最长的单词。用一个可以重复以下示例的主程序来测试这个函数。

```
Enter a line:
All mimsy were the borogoves.↵
The longest word is "borogoves".
```

7. 写一个函数ReverseSentence(line)，把line分成单词并返回由相同单词倒序组成的字符串。函数应该丢弃所有line中的标点符号，并在结果中用一个空格分隔每个单词。例如，主程序读入一行，然后调用ReverseSentence显示结果，可能产生如下输出：

```
Enter a line: double, double toil and trouble:↵
trouble and toil double double
```

8. 扩展scanner.h接口，使它包括在10.2.4节中提到的ReturnUppercaseTokens函数。写一个测试程序通过读入一行数据并且显示每个记号来测试你的扫描器，每个记号占一行。如以下运行示例所示：

```
Enter a line: Hello there.↵
The tokens are:
   "HELLO"
   " "
   "THERE"
```

确保piglatin.c程序不需任何内部改动，就可以使用这个扩展的扫描器模块。

9. 在C程序中，空格和Tab只用来分隔独立的记号而没有其他意义。如果要为C编译器写一个记号扫描器，丢弃由全部空白空格组成的记号是有用的。设计一个scanner.h接口的扩展，允许客户指定空格、Tab和任何其他在ctype.h中的isspace函数返回TRUE的字符被忽略掉。确保你的设计不会影响任何scanner.h接口的任何现有客户。

10. 写一个程序calc.c，实现一个简单的算术计算器。计算器的输入包含由整型常量构成的行，常量由五个C语言中使用的算术运算符（+、-、*、/和%）分隔。对每行输入，程序应该显示每个运算符的运算结果。为了读入单独的值和运算符，你该使用第9题中扩展的扫描器模块来忽略空格。当用户输入一个空行时，程序应该退出。 369

为了降低程序的复杂性，计算器应该忽略C语言的优先级规则，而采用从左到右的运算次序。这样，表达式2 + 3 * 4该得到20而不是C语言中的14。

下面是calc.c的程序的运行示例：

```
This program implements a simple calculator.
When the > prompt appears, enter an expression
consisting of integer constants and the operators
+, -, *, /, and %. To stop, enter a blank line.
> 2 + 2↵
4
```

```
The primes between 0 and 200 are: 2, 3, 5, 7, 11, 13,
17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71,
73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131,
137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191,
193, 197, 199.
```

371

```
/*
 * fill.h
 * ------
 * This file is the interface to an abstraction that enables
 * you to display filled output.  To use the package, you first
 * call SetFillMargin to set an output margin and then make
 * calls to PrintFilledString to display the output.  These
 * functions are described in more detail below.
 */

#ifndef _fill_h
#define _fill_h

#include "genlib.h"

/*
 * Function: SetFillMargin
 * Usage: SetFillMargin(margin);
 * -----------------------------
 * This function sets the fill margin to the value given by the
 * margin parameter.  When strings are displayed using the
 * function PrintFilledString, the output will be broken up
 * into separate lines so that no lines extend past the margin.
 * If no margin is set explicitly, the package uses 65 for the
 * fill margin as a default.
 */

void SetFillMargin(int margin);

/*
 * Function: PrintFilledString
 * Usage: PrintFilledString(str);
 * ------------------------------
 * The string str is displayed on the screen, starting from
 * where the last string left off.  Spaces and newline characters
 * in the string are treated specially.  If a space character
 * appears in an argument, the function may choose to divide
 * the string at that point by moving to the next line.
 * Such division occurs when the next word would not fit
 * within the margin established by SetFillMargin.  A newline
 * character in the string forces the cursor to advance to the
 * beginning of the next line.
 */

void PrintFilledString(string str);

#endif
```

图10-6 fill.h

372

第三部分

复合数据类型

概述

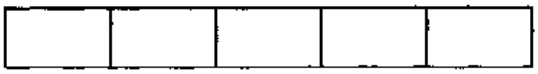
虽然你已经知道程序操作数据，但本书前面部分重点讨论执行操作的控制结构，而不是基本的数据。在第11章～第17章，你将会看到，和控制结构一样，数据结构也是程序设计的中心。通过学习将数据组装成较大的结构，能极大地提高程序设计的能力，并能写出更有用和更激动人心的程序。

1) 数组是有序的。你必须能把数组中的每个分量按顺序排列：这是第一个，这是第二个，等等。

2) 数组是同质的。数组中的每个数值必须有同样的类型。因此你可以定义一个整型数组或是一个浮点型数组，但是一个数组中不能有两种数据类型。

376

直观上，我们可以把数组看成一系列的方框，数组中的每个数值占一个方框。数组中的每一个数值称为一个**元素**（element）。例如，右图代表一个有五个元素的数组。



在C语言中，每个数组有两个基本特性：

- **元素类型**（element type） 存储在数组元素中的数值的类型。
- **数组大小**（array size） 数组所包含的元素个数。

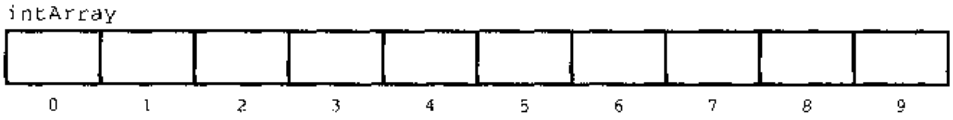
当你在程序中建立一个新的数组时，你必须指明数组的元素类型和数组大小。

11.1.1 数组声明

和C语言中其他的变量一样，数组在使用前必须声明。下面的语法框中给出数组声明的一般形式。例如：

```
int intArray[10];
```

声明了一个名为intArray，且拥有十个元素的数组，每个元素的类型都是int型的。你可以用下面图示的方法描述这种定义，画出10个方框，且命名为intArray：



数组中的每一个元素都由对应的一个叫做**下标**（index）的数值确定。在C语言中，数组的下标从0开始到比数组大小小1的数为止。因此，一个数组有10个元素，那么下标就是0、1、2、3、4、5、6、7、8和9，如上图所示。

语法：数组声明

elementtype arrayname[*size*];

其中：

*elementtype*是数组中每个元素的类型。

*arrayname*是声明为一个数组的变量名。

*size*作为数组一部分的元素数。

在大多数情况下，应该用一个符号化的常量而不是一个具体的数值来指定数组的大小。用一个符号化的常量使得程序员将来可以容易地修改数组的大小。例如，定义数组不能用下列方法：

```
int intArray[10];
```

377

而应该将大小定义成常量：

```
#define NElements 10
```

然后把声明写为

```
int intArray[NElements];
```

11.1.3 一个简单的数组实例

图11-1中给出一个简单的数组操作的实例gymjudge.c。该程序要求用户输入每个裁判员给出的分数，然后显示平均分数。 379

```
/*
 * File: gymjudge.c
 * ----------------
 * This program averages a set of five gymnastic scores.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constants
 * ---------
 * NJudges -- Number of judges
 */

#define NJudges 5

/* Main program */

main()
{
    double gymnasticScores[NJudges];
    double total, average;
    int i;

    printf("Please enter a score for each judge.\n");
    for (i = 0; i < NJudges; i++) {
        printf("Score for judge #%d:  ", i);
        gymnasticScores[i] = GetReal();
    }
    total = 0;
    for (i = 0; i < NJudges; i++) {
        total += gymnasticScores[i];
    }
    average = total / NJudges;
    printf("The average score is %.2f\n", average);
}
```

图11-1 gymjudge.c

使用前一节例子中的数据运行程序gymjudge.c，你会得到如下结果：

```
Please enter a score for each judge.
Score for judge #0:  9.2↵
Score for judge #1:  9.9↵
Score for judge #2:  9.7↵
Score for judge #3:  9.0↵
Score for judge #4:  9.5↵
The average score is 9.46
```

380

11.1.4 改变下标值的范围

在C语言中，每个数组的第一个元素的下标值是0。然而，在许多情况下使用这种设计方式

为了弄清数据在计算机中是如何存储的，先要仔细地了解一下在一个典型的机器中内存系统是如何工作的。计算机的内存系统在1.3节中有所涉及，但我们还是不知道内存是如何工作的。

11.2.1 比特、字节和字

从本质上讲，计算机中所有的数值都是按照被称为比特的基本单元的形式存储的。1个**比特**（bit）精确记录一个数值，它可能取两种状态中的一个。如果把机器中的电路当成一个小的开关，那么可以把这两种状态标识为断开和闭合。单词bit实际上是binary digit的缩写，因此把这两种状态用0和1表示就很顺理成章了，0和1这两个数字是计算机运算所基于的二进制系统所用的两个数字。

因为1个比特包含的信息量极为有限，所以比特本身并没有为存储数据提供一个很方便的机制。为了方便地存储整型和字符型等常见类型的信息，将若干个独立的比特组成一个较大的单元，作为完整的存储单元。这种最小的组合单位被称为**字节**（byte），它足以容纳一个需要8个比特的字符[⊖]。在大多数计算机中，字节也可以组成更大一些的称为**字**（word）的结构，一个字的大小通常要求能够容纳一个整型数据。有些计算机用两个字节组成字，有些计算机用四个字节组成一个字，也有很少的计算机的规定十分特殊。

某台计算机可用的内存容量变化很大。内存大小通常用千字节（kilobyte，KB）和兆字节（megabyte，MB）来衡量。在大多数领域中，前缀kilo和mega分别代表1000和1000 000。然 382
而，对计算机而言，十进制系统与机器的内部结构是不匹配的，因此只是用这些前缀表示接近于原来的数值的2的某个整数幂。因此，在程序设计中，这些前缀真正表示的意思应该是：

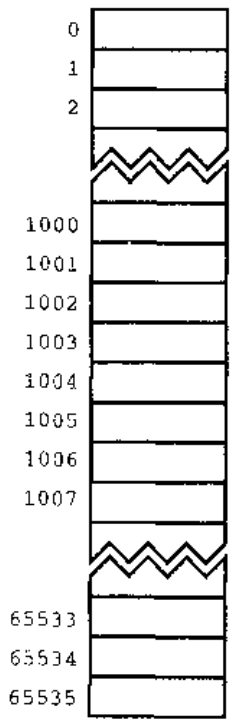
$$\text{kilo(K)} = 2^{10} = 1024$$

$$\text{mega(M)} = 2^{20} = 1\,048\,576$$

一个64KB的计算机（用现在的标准来衡量属于小型的机器）的内存容量是64×1024，即65 536字节。同样，一个8MB的工作站有8×1048576，即8 388 608字节的容量。

11.2.2 内存地址

在存储系统中，每个字节都会有一个数字的**地址**（address）。一般来说，第一个字节在计算机中的地址为0，第二个字节的地址为1，依此类推，直到机器中所含有的字节数为止。例如，你可以用右图表示64KB的计算机的内存。内存中的每一个字节可以容纳一个字符的信息。例如，如果你声明一个字符变量ch，编译器会在当前的函数帧中为这个变量保留一个字节的空间。假设这个字节的地址为1000。如果程序执行语句：




```
ch = 'A';
```

字符'A'的内部表示就会被存储在地址为1000的单元中。由于'A'的ASCII码是65，因此内存的分配结果如下页图a所示。 383

⊖ 虽然几乎在所有的现代计算机中，一个字节由8个比特组成，但ANSI C并没有要求一个字节一定要那么长。仅有的规则是char类型的对象被定义为占据一个字节。因此，一个字节的长度必须足以容纳一个字符代码。

空间。局部变量只有当函数调用时才会分配内存空间。变量本身在分配给这个函数的存储空间中分配，函数的存储空间叫做函数的**帧**（frame）。只要函数在运行，局部变量在帧中的地址始终是不变的。当函数返回时，帧以及它的所有变量都被丢弃，以便让别的函数使用。

如果你声明了一个简单变量，编译器就会为这个变量预留其需要的内存空间，即是sizeof的返回值。如果你声明了一个数组变量，编译器就会为数组元素预留一段连续的内存空间保存组成数组的所有元素的值。根据元素的类型，数组的每个元素可以需要1个字节的内存空间，也可以需要几个字节的内存空间。例如，变量声明：

```
char charArray[20];
```

将会预留20个字节的内存空间，因为每一个字符仅占一个字节。

有些数据类型的每个元素要占有一个以上的字节。例如，假设上述NJudges被设为5，数组中每个元素都是double型的，需要八个字节的内存空间，用下列语句声明的数组scores

```
double scores[NJudges];
```

需要

$$8\text{字节/元素} \times 5\text{个元素} = 40\text{字节}$$

为了存放整个数组，编译器要在当前帧中分配40个字节。如果帧在内存中的开始地址是1000，数组元素的存放情况将如图11-2所示。0号元素将存放在1000～1007，1号元素将存放在1008～1015，依此类推。

当C语言编译器遇到一个选择表达式：

```
scores[i]
```

它通过把下标值和每个元素的大小相乘再加上数组的首地址，来计算出数组元素的地址。在图11-2中，元素的大小是8，数组的首地址是1000。因此，如果i的大小为2，那么scores[i]地址就为

$$2 \times 8 + 1000$$

这个地址是1016，正好与图中scores[2]的地址相匹配。在这种情况下，数组元素的首地址（就是1000）也叫做数组的**基地址**（base address），而找到正确元素所需的调整量（就是2×8，即16）叫做**偏移量**（offset）。

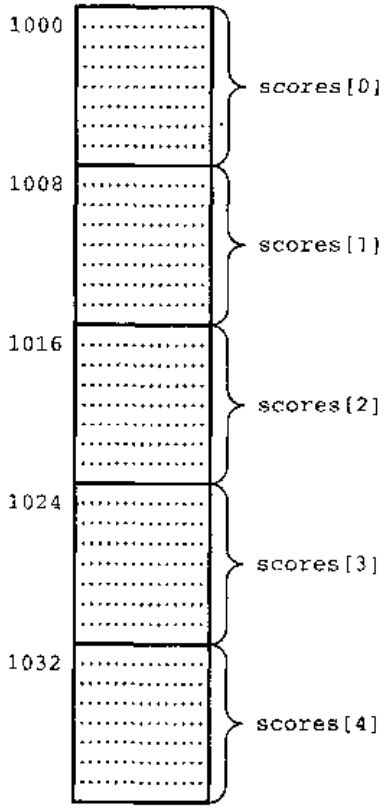


图11-2 scores数组的内存布局

11.2.5 引用超出数组范围的元素

在图11-2中，数组scores有五个合法的下标值：0、1、2、3和4。如果你选择一个下标值超出这个范围的元素，那将会有什么情况发生呢？例如，你要求出表达式

```
scores[i]
```

的值，但这是i的值恰好是5。因为在数组中没有scores[5]，这个结果自然就没有意义。在这种情况下，C语言会如何处理呢？

如果你很幸运使用的是可以防止粗心的程序员犯此类错

函数都小到足以作为一个单元来理解。函数之间通过传递参数来交换信息。如果一个大的程 388
序里面含有数组，那么在分解这个大的程序时就需要函数把整个数组作为参数进行传递。在C语言中，把数组作为参数进行传递和数组在内存中的表示方法有很大关系，有时这不太容易让人理解。在熟悉内部表示法之后，就可以学会数组参数是如何工作的，以及如何有效地使用它们。

将数组作为函数参数的问题可以通过一个简单的例子进行说明。假如让你编写一个名为reverse.c的程序，来实现下面的功能：

1) 读入一串整型数据，直到输入0作为输入结束标志为止。

2) 把这些整型数据逆序排列。

3) 输出经过重新排列后的数据。

下面的运行示例说明了这个程序的操作过程：

```
? 1↵
? 2↵
? 3↵
? 4↵
? 5↵
? 0↵
5
4
3
2
1
```

为了说明数组作为参数的传递过程，重要的是把程序分解成三个函数，分别对应于程序的三个阶段：读入输入数据并将其保存到数组中，对数组元素逆序排列，然后输出结果。用这种方法分解程序的话，主函数就会有如下的结构：

```
main()
{
    int list[NElements];

    GetIntegerArray(list);
    ReverseIntegerArray(list);
    PrintIntegerArray(list);
}
```

（这个程序只是逆序表程序的一个大概的轮廓。最终的版本有不同的参数结构）

然而，这种通用的设计方法会产生两个问题：

1) 在编写的时候，数组元素的个数是用常量Nelements确定的，但在问题说明中，要求元素的个数在用户输入结束标志0之后才能确定。 389

2) 函数GetIntegerArray和ReverseIntegerArray只有在它们改变了参数数组中的数值时才有用。然而，到现在为止，在本书中看到的函数还都不能改变它们的参数值。如果数组和简单变量一样的话，那么整个程序的设计就必须改变了。

上述问题将会在下面几节中具体介绍。

11.3.1 元素个数的通用化

从问题说明中可以看出，程序reverse.c并不能事先确定数组元素的个数，而是让用户输入任意个值，用一个预先设定好的输入结束标志值表示数组结束。这种设计方法对用户来

参数，依此类推。在这个例子中，数组名在主程序中是list，而在子函数中是array。选用不同的名字使你容易区分在主程序中和其他函数中的数组。

因为ReverseIntegerArray函数也需要访问数组数据和有效长度，所以它和PrintIntegerArray有同样的参数。它的函数原型为如下形式：

```
void ReverseIntegerArray(int array[ ], int n);
```

函数GetIntegerArray却有不同的结构。主程序是不可能传递数组的有效长度给GetIntegerArray函数的，在函数GetIntegerArray被调用的时候，主程序是不知道数组的有效长度的。当用户输入数据时，函数GetIntegerArray通过计数确定数组的有效长度，直到用户输入了结束标志值为止。此时，GetIntegerArray必须告诉主程序输入了多少个数值，因此它是返回一个整型数值的函数。

尽管GetIntegerArray函数不需要将数组的有效长度作为参数，但它需要知道数组的分配长度。如果用户输入的数值超过数组的容量，函数GetIntegerArray能报告有错误产生就很重要了。此外，如果GetIntegerArray函数能将输入结束标志值作为参数，那就更通用了，可以调用同样的函数去读入有不同结束标志值的数据串。因此，GetIntegerArray函数原型可以写成：

391

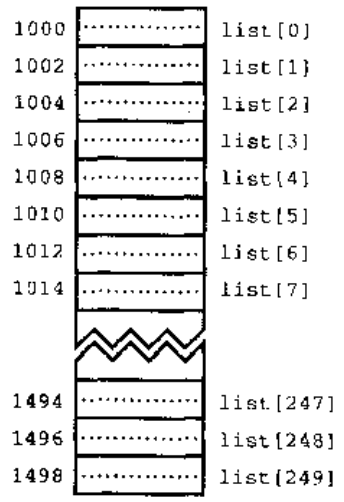
```
int GetIntegerArray (int array[ ],int max,int  sentinel) ;
```

11.3.2 数组参数传递机制

在你实现这些函数之前，必须解决有关数组参数的其他一些问题。当用一个简单变量调用函数时，函数会收到被调用参数的一个拷贝。因为函数是用拷贝进行工作而不是用原来的数值工作，所以函数不会改变调用参数的数值。然而，当一个有数组参数的函数被调用时，形式参数和实际参数之间的关系就会改变。

为了理解数组如何从一个函数传送到另外一个函数，需要考虑数组在内存中的表示方式。假设MaxElements被定义成250，以下的数组声明

```
int list[MaxElements];
```



要求编译器在main的帧中为有250个整型数据的数组分配空间。虽然不可能确切知道在计算机中使用什么地址，我们可以假设分配给数组list的内存起始地址是1000，连续分配250个整型大小的字。如果在你使用的计算机中，整型数据占两个字节，那么数组list的数据将会存储在地址1000～1499中，如右图所示。当你编写一个有数组参数的函数时，通常希望函数直接利用对应于数组的内存中的数值。C语言很好的做到了这一点。当数组作为参数传送给函数时，只有数组的基地址记录在局部帧中。如果你选择在函数范围内定义的局部数组中的一个元素，那么把偏移量加上基地址就会生成在调用数组范围之内的地址。最终结果是在函数头部定义的形式数组参数成为函数调用中实际数组参数的同义词，而不是它的拷贝。

为理解数组操作的细节，可以思考主程序在调用ReverseIntegerArray函数时会有什么情况发生。如果数组含有五个数值1、2、3、4和5，那么对于函数ReverseIntegerArray，帧的初始配置如下图所示：

因为在C语言中，数组参数的行为不同于其他的参数，所以有必要按照下面的规则考虑它们的区别：

> **数组参数规则** 如果调用一个以数组作为形式参数的函数，那么用作实际参数的数组的存储空间被形式参数所共享。改变形式参数中某一元素的值也会改变实际参数数组中的对应元素。

11.3.3 实现函数PrintIntegerArray和GetIntegerArray

为了完成程序reverse.c，需要实现被主程序调用的每一个函数。虽然可以以任何顺序写这些函数，但一般是先写最简单的。在这个例子中，最简单的函数当然是PrintIntegerArray。这个函数的主要任务是遍历并分行输出数组元素。遍历过程只需要一个for循环语句，开始为0，直到遍历完所有的有效元素为止。for循环语句的上限是数组的有效长度n。函数PrintIntegerArray的实现如下：

394

```
static void PrintIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("%d\n", array[i]);
    }
}
```

函数GetIntegerArray的实现有一些复杂，主要是因为你要进行检测以确保用户输入的数据没有超过数组的最大容量。因为C语言没有提供措施避免输入数据超过数组的限制，所以这种检测就至关重要。如果你没有检测且用户输入过多的数据，这些数据就会覆盖其他的重要的信息，使你的程序莫名其妙地不能运行。下面的GetIntegerArray函数的实现就注意检测是否输入过多数据：

```
static int GetIntegerArray(int array[], int max, int sentinel)
{
    int n, value;

    n = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetInteger();
        if (value == sentinel) break;
        if (n == max) Error("Too many input items for array");
        array[n] = value;
        n++;
    }
    return (n);
}
```

一旦实现了GetIntegerArray和PrintIntegerArray两个函数之后，就要在编写ReverseIntegerArray之前对它们进行测试。例如，你可以用下面的主程序对两个函数进行编译：

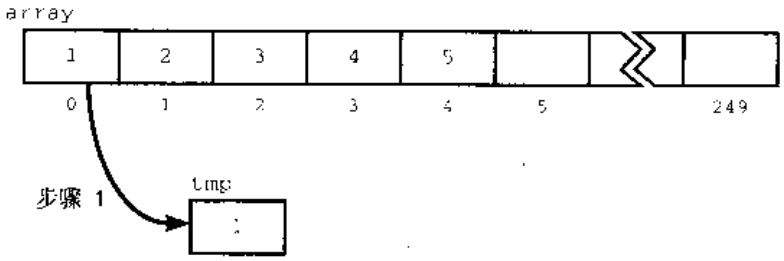
```
main()
{
    int list[MaxElements], n;
```

11.3.5 实现函数SwapIntegerElements

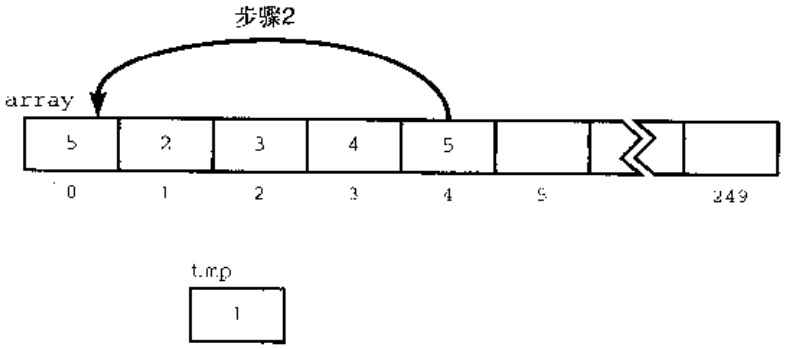
实现函数SwapIntegerElements要比它刚出现的时候复杂一些。你不能把一个元素值赋值给另外一个元素，因为目标元素的初始值会丢失。解决此问题的最简单的办法就是用一个局部变量暂时把此元素值存放起来。如果你保持一个元素值不丢失，你就可以很轻松地对它进行赋值操作了，然后再把暂时存放在局部变量的数值拷贝过来。

假设你想交换下标值为0和4的数组元素。可以分成三步实现这种操作：

1) 把array[0]的值暂时存放到一个临时变量中，如下图所示：

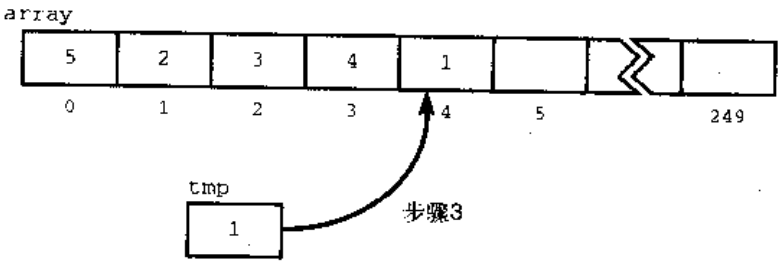


2) 把array[4]中的数值复制到array[0]中，如下图所示：



因为元素array[0]原来的数值已经存放在tmp中，所以没有信息丢失。 397

3) 把tmp中的值赋给array[4]，如下图所示：



这三步是实现函数SwapIntegerElements的基本步骤：

```
static void SwapIntegerElements(int array[], int p1, int p2)
{
    int tmp;

    tmp = array[p1];
    array[p1] = array[p2];
    array[p2] = tmp;
}
```

这个函数补全了reverse.c程序，完整的程序出现如图11-3所示。

```
 * elements, GetIntegerArray generates an error.
 */

static int GetIntegerArray(int array[], int max, int sentinel)
{
    int n, value;

    n = 0;
    while (TRUE) {
        printf(" ? ");
        value = GetInteger();
        if (value == sentinel) break;
        if (n == max) Error("Too many input items for array");
        array[n] = value;
        n++;
    }
    return (n);
}

/*
 * Function: PrintIntegerArray
 * Usage: PrintIntegerArray(array, n);
 * -----------------------------------
 * This function displays the first n values in array,
 * one per line, on the console.
 */

static void PrintIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("%d\n", array[i]);
    }
}


/*
 * Function: ReverseIntegerArray
 * Usage: ReverseIntegerArray(array, n);
 * -------------------------------------
 * This function reverses the elements of array, which has n as
 * its effective size.  The procedure operates by going through
 * the first half of the array and swapping each element with
 * its counterpart at the end of the array.
 */

static void ReverseIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n / 2; i++) {
        SwapIntegerElements(array, i, n - i - 1);
    }
}
```

图 11-3 (续)

入一个空行表示输入结束的时候，程序应该输出一个表，说明在输入数据中各个字母出现的次数。程序的执行可以用如下运行示例说明：

```
This program counts letter frequencies.
Enter a blank line to signal end of input.
Peter Piper picked a peck.↵
of pickled peppers.↵
↵
A     1
C     3
D     2
E     8
F     1
I     3
K     3
L     1
O     1
P     9
R     3
S     1
T     1
```

为了生成上述的字母出现频率表，程序必须逐个字符地搜索输入文本中的每一行。一旦一个字母出现，程序就会更新一个记录此字母出现次数的数值。这个程序值得注意的地方就是要设计一种数据结构，可以记录26个字母出现的次数。

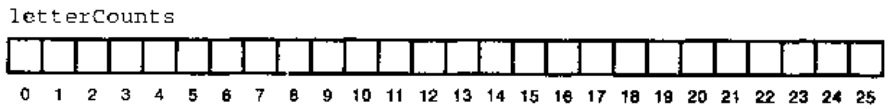
不使用数组，定义26个不同的变量nA，nB，nC依次到nZ也可以解决这个问题，然后用一个switch语句来检查26种情况：

```
switch (toupper(ch)) {
  case 'A': nA++; break;
  case 'B': nB++; break;
  case 'C': nC++; break;
  . . .
  case 'Z': nZ++; break;
}
```

这是一个很长的且重复性的程序。更好的方法就是把26个变量放入一个数组，然后使用字符代码选择数组中合适的元素。每个元素里存放一个整型数值，表示对应于数组中该下标的字母当前出现的次数。如果你调用数组letterCounts，你可以把它声明为： 402

```
int letterCounts[NLetters];
```

Nletters被定义为常数26。这个声明给一个有26个元素的整型数组分配内存空间，如下图所示：



在输入数据中每出现一个字母时，就要增加letteCounts中相应元素的值。寻找需要自增的元素的过程就是用字符运算把一个字母转换成在范围0～25中的整数。这个转换过程由函数LetterIndex完成，该函数的实现如下：

```
int LetterIndex(char ch)
{
    if (isalpha(ch)) {
```

注意，for循环语句的循环范围是从'A'～'Z'，而不是从0～25是很重要的。因为我们用对应于我们问题的下标值，会有利于我们理解for循环语句。同时还应该注意到，在这个例子中也无需检测函数LetterIndex的返回值是否是－1，因为函数DisplayLetterCounts的结构已经确保不会有超出范围的情况发生。

图11-4中程序的其他部分包括逐字扫描输入数据的代码。

```
/*
 * File: countlet.c
 * ----------------
 * This program counts the occurrences of individual letters
 * that appear in text read in from the user.  This program
 * might be useful as a tool in solving cryptograms.
 */

#include <stdio.h>
#include <ctype.h>
#include "simpio.h"
#include "strlib.h"
#include "genlib.h"

/*
 * Constants
 * ---------
 * MaxLines -- Maximum number of input lines
 * NLetters -- Number of letters
 */

#define MaxLines 100
#define NLetters  26

/* Private function declarations */

static void CountLetters(int letterCounts[]);
static void CountLettersInString(string str, int letterCounts[]);
static void RecordLetter(char ch, int letterCounts[]);
static void DisplayLetterCounts(const int letterCounts[]);
static int LetterIndex(char ch);
static void ClearIntegerArray(int array[], int n);

/* Main program */

main()
{
    int letterCounts[NLetters];

    printf("This program counts letter frequencies.\n");
    printf("Enter a blank line to signal end of input.\n");
    ClearIntegerArray(letterCounts, NLetters);
    CountLetters(letterCounts);
    DisplayLetterCounts(letterCounts);
}

/*
 * Function: CountLetters
```

图11-4 countlet.c

```
 * ----------------------------------------
 * This function displays the letter frequency table, leaving
 * out any letters that did not occur.
 */

void DisplayLetterCounts(const int letterCounts[])
{
    char ch;
    int num;

    for (ch = 'A'; ch <= 'Z'; ch++) {
        num = letterCounts[LetterIndex(ch)];
        if (num != 0) printf("%c  %4d\n", ch, num);
    }
}

/*
 * Function: LetterIndex
 * Usage: index = LetterIndex(ch);
 * ------------------------------
 * This function converts a character into the appropriate index
 * for use with the letterCounts array.  In this implementation,
 * LetterIndex converts characters in either case to an integer
 * in the range 0 to 25.  If ch is not a valid letter, LetterIndex
 * returns -1.  Clients should check for a -1 return value unless
 * they are able to guarantee that the argument is a letter.
 */

int LetterIndex(char ch)
{
    if (isalpha(ch)) {
        return (toupper(ch) - 'A');
    } else {
        return (-1);
    }
}

/*
 * Function: ClearIntegerArray
 * Usage: ClearIntegerArray(array, n);
 * -----------------------------------
 * This function sets the first n elements in the array to 0.
 */

static void ClearIntegerArray(int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        array[i] = 0;
    }
}
```

图 11-4 (续)

11.5 数组的静态初始化

数组变量可以和其他变量一样被声明为局部变量或全局变量。为了避免使用全局变量的

对于一个数组a，可以通过下面的表达式计算其中的元素的个数：

```
sizeof a/ sizeof a[0]
```

用自然语言描述，这个表达式表示用数组中的第一个元素的大小去除整个数组的大小。因为数组中的每个元素的大小都是一样的，不管元素是什么类型，上述表达式的结果就是数组中元素的个数。因此，你可以用下列表达式初始化一个变量bigCities保存数组中的城市数：

```
static int nBigCities-sizeof bigCities/ sizeof bigCities[0];
```

11.5.3 初始化数组和标量类型

根据第9章中的定义，标量类型可以用在任何整型数据出现的地方。因此可以使用任何标量类型值作为数组的下标值。这样就增加了标量类型的功能，也简化了它们的应用。

作为一个实例，初始化的数组可以很容易把枚举类型的数据转化成与它们相对应的名字，这种操作在C语言中是不直接支持的。例如，如果你想用FALSE和TRUE两个名字来输出布尔类型的数据，你可以声明数组booleanNames如下：

```
static string booleanNames = {"FALSE", "TRUE"};
```

409

然后使用printf语句：

```
printf("flag-%s\n', booleanNames[flag]);
```

因为布尔类型在genlib.b中被定义成：

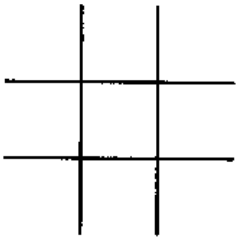
```
typedef enum {FALSE, TRUE} bool;
```

常量FALSE的内部值为0，TRUE的内部值为1。使用这些数值作为数组booleanNames的下标值就可以生成对应于布尔型值的字符串。这种情况也同样适用于连续枚举类型。

11.6 多维数组

在C语言中，数组元素可以是任何类型的。特别是，数组的元素又是数组。这种数组的数组就叫做**多维数组**（multidimensional array）。多维数组中最常见的就是二维数组，它常用于表示分割成行和列的矩形的数据结构。这种二维结构的类型也叫做**矩阵**（matrix）。三维或是多维的数组在C语言中虽然是合法的，但是很少出现。

作为二维数组的例子，假设要在程序中表述tic-tac-toe这种游戏。你可能知道，tic-tac-toe这种游戏是在一个分成三行三列的面板上进行的，如右图所示。参加游戏者轮流在这些空的方块中填入X和O，尽量使相同的符号在水平方向、垂直方向或者对角线上排成一行。



为了描述这种游戏的面板，最好的选择就是使用一个三行三列的二维数组。虽然也可以使用枚举类型来表示在方块中可能出现的三种情况——空格，X和O，但是用char型数据作为元素类型并用字符' '，'X'和'O'表示方块的三种情况可能会更简单一些。这种游戏面板的声明可以写为：

```
char board[3][3];
```

有了上述声明，你就可以通过给出两个下标值来确定面板方块中的字符，一个下标确定行号，一个下标确定列号。在这种表示方法中，行号和列号的范围是0～2，面板中各个方块有如下名字：

410

```
static void DisplayBoard(char board[ ][3])
```

程序可以以同样的方式工作，因为对C语言来讲，知道数组的基地址和第二个下标的大小就可以确定数组board中每一个元素的地址，而不管行数是多少。然而，略去第一个下标值使声明很不对称，因此，一般情况下，在声明多维数组参数时，每一个下标的范围都应该写上。

11.6.2 初始化多维数组

和初始化一维数组一样，可以对多维数组进行静态初始化。为了强调多维数组的整体结构，用来初始化内部数组的数值都被放到一组花括号中。例如：

```
static double identityMatrix[3][3] = {
    { 1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 }
};
```

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

412

声明了一个3×3的浮点数矩阵，并用右图的数据对它进行初始化。

这个特殊的矩阵在数学中经常出现，被称为**单位矩阵**（identity matrix）。

和参数的情况一样，静态初始化的多维数组的声明必须指定所有下标值的范围，第一个下标值除外，因为它可以通过计算初始值个数确定。但如果作为传送参数，最好在声明多维数组时明确指定每个下标值的范围。

关于对多维数组进行静态初始化的更多的实例将在第12章中介绍。

小结

本章介绍了两个截然不同但又相关的概念：数据在内存中的表示和称之为数组的特殊数据结构。有关内存和数据表示，要掌握的基本知识有：

- 每个变量都存储在内存中的某个地方，因此都有一个地址。
- 一旦被分配，变量的地址不再改变，而它的内容可以改变。
- 根据所存储的数据类型，变量需要不同大小的内存。同时随着计算机的不同，相同的变量类型需要的存储空间也有所不同。在某个计算机上编译过的程序中，可以用运算符sizeof来确定存储某一变量值需要的字节数。

对于数组来说，本章讨论以下几项重要内容：

- 数组通常用来存储具有同一数据类型并且按顺序排列的一组数据。数组的每个成员称为*元素*，通常用下标值来标识。在C语言中，所有数组的下标都是从0开始的。
- 当声明一个数组变量时，必须指定数组的大小，而且它必须是一个常量。对于数据量可变的应用来说，常用的方法是在数组中分配足够的空间，使之能保存最大的数据量，而使用时只使用其中的一部分。元素的最大个数称为数组的*分配长度*，实际使用的元素个数称为*有效长度*。 413
- 引用数组中的元素的过程称为*选择*（selection），一般用数组名后跟一个括在方括号内的表达式表示。
- 在内存中，数组元素通常是连续存储的。数组第一个元素的地址称为*基地址*。选择特定元素所需的调整量称为该元素的*偏移量*。
- 当把一个数组作为参数传递给函数时，对应的形式参数被初始化以保存调用数组的基地址。在函数内部，所有涉及到对形式参数进行的操作都转化为对该数组进行操作，因此数组和函数的参数实质上是等同的，而其他的参数类型只是被调参数的拷贝。

17. 本章介绍的确定一个静态初始化数组的元素个数的习语是什么？ 415

18. 在第9章对枚举类型的讨论中，类型directionT的定义如下：

```
typedef enum {North, East, South, West} directionT;
```

给定一个directionT型变量dir，如何用静态初始化数组将方向的名字显示在屏幕上？

19. 什么是多维数组？

20. 假如某数组的基地址是1000，int类型的值在内存中占2个字节，请用图表示下面数组声明中各元素的地址：

```
int rectangular [2][3];
```

21. 为表示8×8的棋盘中每一个地方对应什么棋子，我们将定义什么变量呢？假设该棋盘中每个地方对应的元素是以下元素中的一个：

K	white king	k	black king
Q	white queen	q	black queen
R	white rook	r	black rook
B	white bishop	b	black bishop
N	white knight	n	black knight
P	white pawn	p	black pawn
-	empty square		

如何静态初始化这个数组，使它保存标准的开始位置。

r	n	b	q	k	b	n	r
p	p	p	p	p	p	p	p
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

程序设计练习

1. 写一个程序，用运算符sizeof显示存储各基本数据类型所需要的字节数。在我用过的某台机器上，程序执行的结果可能如下面运行示例所示： 416

```
Values of type char require 1 byte.
Values of type int require 2 bytes.
Values of type short require 2 bytes.
Values of type long require 4 bytes.
Values of type float require 4 bytes.
Values of type double require 12 bytes.
```

因为随着计算机的不同，各数据类型对应的存储空间大小会有所不同，因此输出的结果也会有所不同。

2. 由于各裁判打分时存在某些主观因素，因此在计算平均值时通常去掉一个最高分，去掉一个最低分。写一个程序，读入7个裁判所打的分数，去掉一个最低分，去掉一个最高分，求剩余5个元素的平均值。

3. 在统计学中，一组数据通常有一个**分布**（distribution）。统计分析的主要的目的就是找到一种方法将整个数据集压缩为统计特征，它表示了总体特征。最常见的统计量就是**均值**（mean），也就是简单的平均。对于序列$x_1, x_2, \cdots, x_n$，通常用$\bar{x}$表示其均值。

编写一个函数Mean(array, n)，返回有效长度为n的double型数组的平均值。将该函数加入到图11-1中的gymjudge.c中，测试其是否正确。

4. 另外一个常见的统计量是**标准差**（standard deviation），它表示数列中各值偏离平均值的程度。为计算$x_1, x_2, \cdots, x_n$的标准差，基本步骤如下：

A	•—	J	•———	S	•••
B	—•••	K	—•—	T	—
C	—•—•	L	•—••	U	••—
D	—••	M	——	V	•••—
E	•	N	—•	W	•——
F	••—•	O	———	X	—••—
G	——•	P	•——•	Y	—•——
H	••••	Q	——•—	Z	——••
I	••	R	•—•		

我们很容易在程序中声明一个含有26个元素的数组，存储26个字母的Morse码。

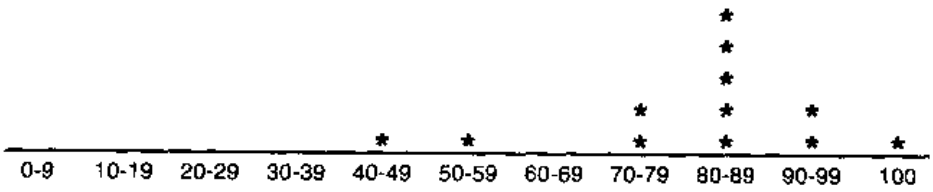
写一个程序，从用户处读入一个字符串，将字符串中每个字母转换成相应的Morse码，用句号表示点，用破折号表示横。用换行符来表示单词之间的空格，而忽略其中的其他标点符号。你的程序应该产生下列运行示例：

```
This program translates a line into Morse code.
Enter English text: What hath God wrought↵
.-- .... .- -
.... .- - ....
--. --- -..
.-- .-. --- ..- --. .... -
```

7. 柱状图是一种显示每个区间内出现的数据的频率的图，例如，对下面的一组成绩：

100，95，47，88，86，92，75，89，81，70，55，80 419

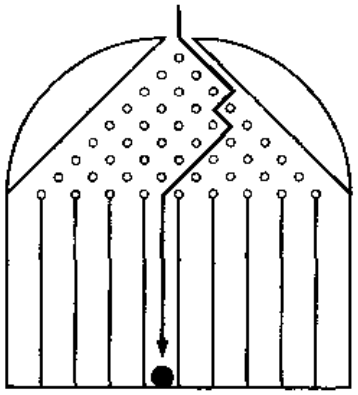
我们可以得到下述的柱状图：



柱状图中的星形表示有一个数据在40～50之间，一个数据在50～60之间，5个数据在80～90之间，依此类推。

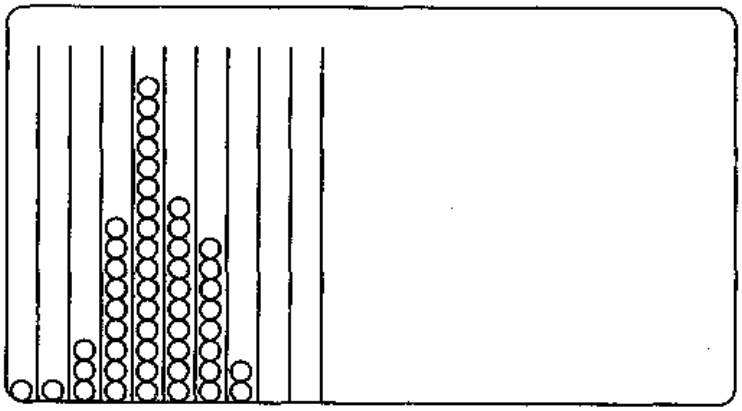
我们用计算机产生柱状图时，显示成以下形式比较容易：

```
0-9   |
10-19 |
20-29 |
30-39 |
40-49 | *
50-59 | *
60-69 |
70-79 | **
80-89 | *****
90-99 | **
100   | *
```



写一个程序，模拟丢下50个弹球，最下面有10个沟道的情况，如下图所示。你编写的程序可以用第7章中的图形库显示图形化的结果。当每一个弹球落地时，程序应在相应的地方画一个表示弹球的图。

下图是50个弹球落下后的一种可能情形：



421
~
422

注意，弹球可能会集中在中间一个沟道中，原因是弹球有更多可能的路径到达中间的沟道。例如要到达最左边的沟道，那么弹球9次都必须从销的左边掉下。相比之下，弹球有很多种路径从顶部到达中间两个沟道，因为左右掉下的顺序可以发生改变而最终它还会到达中间的两个沟道。一般来讲，随机过程最终终止于某个特定状态的可能性取决于到达这个状态的路径数。

11. 用我们在11.6节中给出的tic-tac-toe画板的声明，写一个谓词函数`IsWinningPosition (board, player)`，如果任一个游戏者不管是以`'X'`还是以`'O'`赢得游戏，那么返回`TRUE`。赢的位置是指水平方向、垂直方向或者对角方向的三个元素具有同一字符。编写一个主程序测试函数的功能，读入当前面板数组的内容，然后报告哪个游戏者获得胜利，如下例所示：

```
This program tests the IsWinningPosition function.
Enter the state of the board, row by row.

XOX↵
-XO↵
X-O↵

X has won
```

第12章 查找和排序

Had I been present at the creation, I would have given some useful hints for the better ordering of the universe.

— Alfonso the Wise, thirteenth century

学习目标

- 能够用线性查找和二分查找在数组中找到相应的数据。
- 了解各种算法在效率上的差异。
- 了解一种称为选择排序的简单的排序算法的实现。
- 认识到选择排序算法的效率取决于数据项的个数。

在第11章中，你已经学习了基本的数组操作并且了解了数组在各种应用里如何使用。但是，在第11章中有两个重要的数组操作没有介绍，而这两个重要的操作将在本章中介绍。这两个基本操作就是：

- **查找** 从数组中找出特定元素的过程。
- **排序** 重新排列数组的元素使它们按某一事先确定的顺序存储的过程。

由于查找和排序与数组关系密切，因此从某种意义上说，本章是对数组讨论的继续。但是本章还有另外一个主题，这个主题不仅和第11章有关系，同时和第6章讨论的算法也有关系。由于查找和排序有各种不同的方法，而这些方法的效率又存在很大差异，从而出现了很多有趣的算法问题。

12.1 查找

在前面几章中，我们已经遇到过执行查找的函数，虽然这些函数没有出现在介绍数组的内容中。例如，在piglatin.c程序中（参见图10-5）出现的函数FindFirstVowel的功能就是在一个字符串中查找第一个元音字母。该函数的实现为：

```
int FindFirstVowel(string word)
{
    int i;

    for (i = 0; i < StringLength(word); i++) {
        if (IsVowel(IthChar(word, i))) return (i);
    }
    return (-1);
}
```

该实现采用简单直接的算法来解决查找问题。从字符串的第一个字母开始，该函数依次检查每一个字母，如果该字母是元音字母，那么FindFirstVowel返回该字母在原字符串中的下标。如果在整个字符串中都没有发现元音字母，那么FindFirstVowel将返回值－1告诉客户没有元音字母出现。

12.1.1 在整数数组中查找

你可以利用以上方法在一个数组中查找某个数据项。例如，函数FindIntegerInArray

```
/* Main program */

main()
{
    int value, index;

    printf("This program looks up names of U.S. coins.\n");
    printf("Enter coin value: ");
    value = GetInteger();
    index = FindIntegerInArray(value, coinValues, nCoins);
    if (index == -1) {
        printf("There is no such coin.\n");
    } else {
        printf("That's called a %s.\n", coinNames[index]);
    }
}

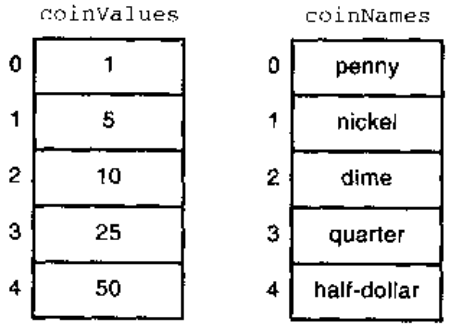
/*
 * Function: FindIntegerInArray
 * Usage: index = FindIntegerInArray(key, array, n);
 * ------------------------------------------------
 * This function returns the index of the first element in the
 * specified array of integers that matches the value key.  If
 * key does not appear in the first n elements of the array,
 * FindIntegerInArray returns -1.
 */

static int FindIntegerInArray(int key, int array[], int n)
{
    int i;

    for (i = 0; i < n; i++) {
        if (key == array[i]) return (i);
    }
    return (-1);
}
```

图 12-1 (续)

FindIntegerInArray函数返回数组coinValues中相应硬币值对应的下标。如果硬币是美国铸造的五种标准硬币之一的话，返回的下标再用来在coinNames数组中选择相应的元素，也就是得到硬币的名称。这两个数组是通过以下原则联系在一起的：硬币的价值的下标和对应的硬币名称的下标是相同的，如右图所示。用相同的下标位置来存储相关的数据称为**并联数组**（parallel array）。



428

12.1.2 关于查找的另一个更复杂的例子

在开始讨论不同查找算法的时候，我们先介绍一个更复杂的查找的应用，而它将就刚才提到的问题做出更容易理解的描述。假设你要在程序中表示一张表（如下所示），该表给出各城市之间的距离[⊖]（用英里表示）：

⊖ 数据来源：*Rand McNally Road Atlas 1994*, New York: Rand McNally。

```
    "San Francisco",
    "Seattle",
};
```

现在已经有了数据，接下来要考虑的就是如何写一个程序读入两个城市的名字，显示它们之间的距离，运行结果如下所示：

```
This program looks up intercity mileage.
Enter name of city #1: San Francisco↵
Enter name of city #2: Boston↵
Distance between San Francisco and Boston: 3128 miles.
```

430

程序必须执行以下步骤：

1) 以字符串的形式读入两个城市的名字。

2) 在数组cityTable中找到该城市名字所对应的下标位置。

3) 用下标值在数组mileageTable中找到对应的值，这个值就是两城市之间的距离。

假设你能够实现一个在字符串数组中查找的函数FindStringInArray，那么程序的其余部分可以直接写成：

```
main()
{
    int city1, city2;

    printf("This program looks up intercity mileage.\n");
    city1 = GetCity("Enter name of city #1: ");
    city2 = GetCity("Enter name of city #2: ");
    printf("Distance between %s", cityTable[city1]);
    printf(" and %s:", cityTable[city2]);
    printf(" %d miles.\n", mileageTable[city1][city2]);
}

static int GetCity(string prompt)
{
    string cityName;
    int index;

    while (TRUE) {
        printf("%s", prompt);
        cityName = GetLine();
        index = FindStringInArray(cityName, cityTable, NCities);
        if (index >= 0) break;
        printf("Unknown city name -- try again.\n");
    }
    return (index);
}
```

12.1.3 线性查找

以上程序中唯一缺少的是函数FindStringInArray的代码。如果遵循实现FindIntegerInArray的逻辑，那么只需要改变参数类型，在循环中使用StringEqual比较字符串的值就可以了。相应的代码和注释见图12-2。

San Francisco所在位置一定是在Houston的后面，因为这个数组是按照字母顺序排列的。因此 433
你可以立即排除下标值从0～5的城市名，留下下列这些位置：

cityTable

0	~~Atlanta~~
1	~~Boston~~
2	~~Chicago~~
3	~~Denver~~
4	~~Detroit~~
5	~~Houston~~
6	Los Angeles
7	Miami
8	New York
9	Philadelphia
10	San Francisco
11	Seattle

仅仅通过上面的一步操作，你已经删去了表中一半的城市名。事实上，更重要的是，在下面的查找中你可以采用和上面相同的方法。如果San Francisco在CityTable中的话，那么它的位置一定是在6～11之间。用上面提到的方法我们再计算该范围的中心：

$$\frac{6+11}{2}$$

由于是整数运算，因此上面表达式的值是8。而同样由字母顺序可知，San Francisco一定在New York的后面，那么又排除了三个城市。

cityTable

0	~~Atlanta~~
1	~~Boston~~
2	~~Chicago~~
3	~~Denver~~
4	~~Detroit~~
5	~~Houston~~
6	~~Los Angeles~~
7	~~Miami~~
8	~~New York~~
9	Philadelphia
10	San Francisco
11	Seattle

434

当你在剩下的范围中再找中间位置时，你发现San Francisco的下标是11，因此整个搜索过程只需要三步。

首先在已排序的数组中查找中间元素，然后根据这个元素的值确定下一步将在哪个部分

```
 * range and compares the element at that index position to the
 * key.  If there is a match, the function returns the index.
 * If the key is less than the string at that index position, the
 * function searches in the first half of the array; if the key is
 * larger, the function searches in the second half of the array.
 */

int FindStringInSortedArray(string key,
                            string array[],
                            int n)
{
    int lh, rh, mid, cmp;

    lh = 0;
    rh = n - 1;
    while (lh <= rh) {
        mid = (lh + rh) / 2;
        cmp = StringCompare(key, array[mid]);
        if (cmp == 0) return (mid);
        if (cmp < 0) {
            rh = mid - 1;
        } else {
            lh = mid + 1;
        }
    }
    return (-1);
}
```

图 12-3（续）

因此，使用线性查找算法查找由N个元素组成的数组需要N次的比较，而使用二分查找算法只需要$\log_2 N$次比较就可以了。

用数学形式表示某些算法的相对效率是定量评价这些算法效率的非常有用的手段。然而对大多数人来说，这种公式仍然不能给人们一个直观的感觉，到底哪个算法更好。那么，我们可以看一些数据。右边的表给出了不同的N值和它相对应的最精确的$\log_2 N$的整数值。从表中的数据可以看出，对于小的数组，两种算法都能很好地完成搜索任务。然而，如果数组的元素个数为1 000 000 000时，在最坏的情况下线性查找算法需要1 000 000 000次的比较才能查找完毕；而二分查找算法最多只需要30次的比较就能查找完毕。很明显，比较次数的锐减表示算法效率大幅增加。

436

N	$\log_2 N$
10	3
100	7
1000	10
1 000 000	20
1 000 000 000	30

二分查找算法唯一的问题就是数组必须是严格地按照某个顺序排列的。如果不是，则必须采用线性查找算法。或者自己先把该数组按要求进行排序。对数组进行排序是比查找更具挑战性的工作，接下来的几节中将介绍这方面的知识。

12.2 排序

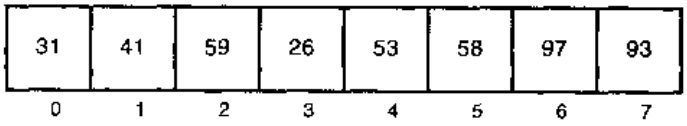
在大多数的商业应用中，计算机通常用来处理一些很简单的工作，例如求某一个数据序列的数值之和或者求它们的平均值，这正好是我们在前几章中已经解决的问题。然而，在某些商业性的程序设计过程中，所要用到的重要操作要更复杂一些。在这些复杂的操作中，最重要的就是**排序**（sorting），它是把一个表（通常表示为数组）按某一次序重新排列。例如，

而实现相应的sort.c程序并不像想象中的那么简单，尤其是当你想找到一种高效率的方法对数据进行排序的时候，情况更是如此。与计算机科学领域内出现的很多问题一样，这里有很多可用的算法。在高级计算机科学教材中，可能需要好几周的时间学习各种不同的排序方法，而每一种算法都有各自的优点和缺点。而在目前的学习阶段，最好从一个排序算法开始，把它的每一个细节都理解清楚。

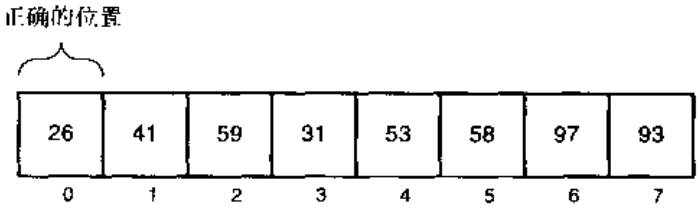
12.2.2 选择排序算法

在众多的排序算法中，最容易理解的一种算法就是**选择排序**（selection sort）算法。当应用选择排序算法时，每次将其中的一个元素放在它最终要放的位置。第一步是找到整个数组中的最小的元素并把它放在它最终要放的位置上，即数组的起始位置，第二步是在剩下的元素中找最小的元素并把它放在第二个位置上。对整个数组重复这个过程，最终将得到按从小到大顺序排列的数组。

为了理解选择排序算法，我们以下面这个数组作为例子：



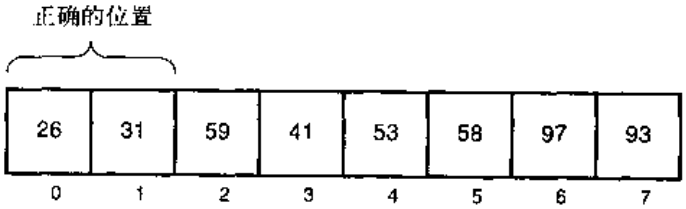
由于该组数中最小的元素值是26，它在数组中的位置是3，因此需要将它移动到位置0。和图11-3中的reverse.c程序一样，我们不希望丢失位置0中的原始值，因此最简单的办法是交换位置0和位置3的值。经过交换得到的新数组如下：



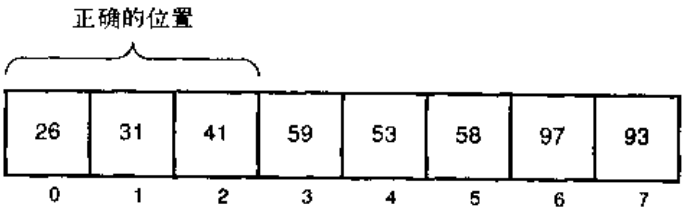
经过交换，位置0的值就是该数组的最小值，符合最终的排序要求。

现在，可以处理表中剩下的数据。下一步是用同样的策略正确填入数组的第2个位置中的值。最小的值（除了已经被正确放置的26外）是31，现在它的位置是3。如果将它的值和下标位置为1的元素的值进行交换，可以得到下面的状态，前两个元素是正确的值：

438 ~ 439



在下一个周期中，再将下一个最小值（应该是41）和位置2中的元素值进行交换：



的SwapIntegerElements是同一个函数，在此我们就不再赘述。当你写完一个实现某个通用操作的函数时，最好把它放在手边，以便以后使用。有经验的程序员通常尽可能重用已有的代码，这样就可以省去由编写和调试这部分程序带来的麻烦。 441

只需要将SwapIntegerElements函数的代码拷贝到选择排序算法中即可完成整个选择排序算法。sort.c实现的完整代码参见图12-5。

12.2.3 选择排序效率的评估

选择排序算法有很多优点。首先，它的算法很容易理解；其次，它解决了排序这个问题。但是，还存在其他一些更有效的排序算法。但是，效率最高的排序算法需要很高的技巧，超出了你现在所掌握的程序设计知识。由于这个原因，我们将在第17章中再讨论更有效的排序算法。

```
/*
 * File: sort.c
 * ------------
 * This file implements the sort.h interface using the selection
 * sort algorithm.
 */

#include <stdio.h>
#include "genlib.h"
#include "sort.h"

/* Private function prototypes */

static int FindSmallestInteger(int array[], int low, int high);
static void SwapIntegerElements(int array[], int p1, int p2);

/*
 * Function: SortIntegerArray
 * --------------------------
 * This implementation uses an algorithm called selection sort,
 * which can be described in English as follows.  With your left
 * hand, point at each element in the array in turn, starting at
 * index 0.  At each step in the cycle:
 *
 * (1)  Find the smallest element in the range between your left
 *      hand and the end of the array, and point at that element
 *      with your right hand.
 *
 * (2)  Move that element into its correct index position by
 *      switching the elements indicated by your left and right
 *      hands.
 */

void SortIntegerArray(int array[], int n)
{
    int lh, rh;

    for (lh = 0; lh < n; lh++) {
        rh = FindSmallestInteger(array, lh, n-1);
        SwapIntegerElements(array, lh, rh);
    }
}
```

图12-5 sort.c

达式将它转化成以秒来计算的时间形式：

```
(double) clock ( ) /CLOCKS_PER_SEC
```

如果把开始时间和结束时间分别存储于变量start和finish中，可以用下列代码计算执行一个操作所需的时间：

```
double start, finish, elapsed;

start = (double) clock ( ) / CLOCKS_PER_SEC;
. . . 执行某些计算 . . .
finish = (double) clock ( ) / CLOCKS_PER_SEC;
elapsed = finish - start
```

你也可以用上述方法计算图12-5中的选择排序算法的运行时间。对大小不同的数组调用SortIntegerArray函数所需要的时间见表12-1。在这张表中，*N*表示数组中元素的个数，而“运行时间”列表示使用选择排序算法对相应的数组进行排序所需要的时间（时间单位为毫秒）。

这张表显示了一个有趣的现象。当*N*很小时，选择排序算法需要的时间很少，但随着*N*的增大，执行选择排序算法需要的时间显著增加。举个例子来说，如果数组包含30个值，SortIntegerArray排序这个数组需要1毫秒，当达到800个值时，需要的时间超过0.5秒。而商业应用通常需要对10 000、100 000甚至更多的数进行排序。对这种规模的数组，选择排序算法的速度将慢得惊人。

442
~
444

表12-1 选择排序算法的运行时间（以毫秒为单位）

N	运行时间	*N*	运行时间
10	0.13	100	9.67
20	0.33	200	37.33
30	1.00	400	146.67
40	1.47	800	596.67
50	2.40		

12.2.5 选择排序的算法分析

要理解为什么程序执行的时间会如表12-1所示，首先我们要明白算法的工作原理。考虑表12-1中选择排序的时间数据，当*N*为50时，执行算法需要2.4毫秒，而当*N*为100时，需要的执行时间为9.67毫秒，差不多是*N*为50时需要时间的4倍。我们从表中其他部分也可以得到这个结论：数组的元素个数增加一倍，那么所需要的时间是原来的4倍。我们把这个排序算法称为具有**平方律**（quadratic）的性质，即执行时间和输入数组大小的平方成正比。

选择排序算法具有平方律性质的事实其实并不奇怪，你只要想一下算法的执行过程就可以理解。在对一个具有8个元素的数组进行排序时，该算法需要执行8次外层for循环。第一个循环周期找出8个数中的最小值，下一个循环周期则在剩下的7个元素中找到最小值，依次类推。程序执行的操作次数和该数组的个数成正比，对上面的例子，需要执行的操作的次数为：

$$8+7+6+5+4+3+2+1$$

更一般的情况，假如数组有*N*个数，选择排序需要的时间与下面的和成正比，即：

$$N+N-1+N-2+\cdots+3+2+1$$

而前*N*个数之和我们可以通过数学公式得到更简洁的表达式，即：

$$N+N-1+N-2+\cdots 3+2+1=\frac{N^2+N}{2}$$

前两个色带的颜色代表阻抗的前两位，第三个色带表示10的幂次数，它要和前两位相乘。例如，当颜色依次是黄-紫-橙，那么它对应的阻抗是：

$$47 \times 10^3$$

即47 000Ω。

447

编写一个程序，读入电阻上三个色带的颜色，输出相应的阻抗值，如下面运行示例所示：

```
This program interprets the resistor color code.
Color of first band: Yellow↵
Color of second band: Violet↵
Color of third band: Orange↵
Resistance = 47000 ohms.
```

2. 编写一个和用户猜数字的程序guestnum.c，假设这些用户是小学的学生。首先由用户确定数字，然后回答计算机的一系列问题，直到最终计算机确定这个数字为止。下面的运行示例显示了当用户确定的数字是17时的执行结果：

```
Think of a number between 1 and 100 and I'll
guess it.
Is it 50? no↵
Is it less than 50? yes↵
Is it 25? no↵
Is it less than 25? yes↵
Is it 12? no↵
Is it less than 12? no↵
Is it 18? no↵
Is it less than 18? yes↵
Is it 15? no↵
Is it less than 15? no↵
Is it 16? no↵
Is it less than 16? no↵
Is it 17? yes↵
I guessed the number!
```

3. 写一个谓词函数IsSorted(array, n)，它的参数是数组名和有效长度，如果数组按照非递减次序排列，返回TRUE。

4. 扩展图12-4和图12-5给出的sort.h和sort.c文件，使其不仅输出SortIntegerArray函数，同时还输出一个函数Alphabetize，它对一个字符串数组按词典顺序排序。

5. 在第11章中，习题第3题和第4题要求编写一个程序来计算两个常见的统计量：平均值和标准差。而另一个常用的统计量就是**中值**（median），也就是把数据按照从小到大的顺序排列后处于中间位置的值。如果数组中元素的个数为偶数，那么就不存在中间元素，此时标准的做法是取中间两个元素的平均值。

448

写一个函数Median(array, n)返回浮点数数组的中值。在写程序时不能假设这个数组是排好序的，因此在程序运行时可能改变元素的次序。

6. 除了中值和平均值，一个分布中最有代表性的第三个统计量是表示数组中出现频率最多的元素的值，例如在数组

9. 很多算法问题在它们的解决方案中都和排序有关。例如，你可以根据随机键值“排序”，以打乱一个数组。一种方法就是使用选择排序算法，然后把找到最小值的位置改成选择一个随机位置。该结果是一个打乱算法，其中各种可能的输出是等概率的。

编写一个程序shuffle.c，通过随机排序输出1～52之间的整数。

10. 入门课程中的一个最著名的算法是所谓的荷兰国旗问题，最初由Edsger Dijkstra提出的。假设一个有n个元素的数组，该数组的每个元素是字符——'R'、'W'或'B'，表示荷兰国旗上的颜色之一。开始时，数组中的值是乱七八糟的，如下所示：

R	B	W	W	B	B	R	W	W	R	R	W	R	B	W
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

你需要做的是重新对以上数组进行排序使其按照荷兰国旗的顺序排序，即所有的红色后面都是白色，而所有白色后面都是蓝色。 450

研究解决这个问题的运行示例，找出相应算法：

```
Initial state:
R B W W B B R W W R R W R B W
Swapping positions 1 and 14
R W W W B B R W W R R W R B B
Swapping positions 4 and 13
R W W W B B R W W R R W R B B
Swapping positions 4 and 12
R W W W R B R W W R R W B B B
Swapping positions 1 and 4
R R W W W B R W W R R W B B B
Swapping positions 5 and 11
R R W W W W R W W R R B B B B
Swapping positions 2 and 6
R R R W W W W W W R R B B B B
Swapping positions 3 and 9
R R R R W W W W W W R B B B B
Swapping positions 4 and 10
R R R R R W W W W W W B B B B
```

编写一个程序对任一随机构造的初始数组实现这个算法。

11. 除了在现阶段比较容易理解的选择排序算法外，还有很多其他的排序算法。但是，这些算法在性能上与选择排序算法相比并没有什么显著优点。即使如此，编写这些算法对我们练习数组的使用是很有帮助的。

例如，你也可以应用以下方法对一个整型数组从小到大进行排序。从数组的起始位置开始，检查数组的每一个元素。对邻近的两个元素进行比较，如果这两个数按照从小到大顺序排列，那么继续向下执行，如果这两个元素不是按照从小到大的顺序排列，那么就将它们的位置进行交换。在任何一种情况下，都移到下一对数据。当在表中移动时，每对数据会有重复。第一对中的第二个元素成为下一对中的第一个元素。重复上述操作直到检查完所有元素，即不再需要交换任意两个元素。这个算法被称为**冒泡排序**（bubble sort），因为元素似乎是“浮到”了它最终的位置。

用上述所说的冒泡法排序重新实现函数SortIntegerArray。

第13章 指　针

> Orlando ran her eyes through it and then, using the first finger of her right hand as pointer, read out the following facts as being most germane to the matter.
>
> —Virginia Woolf, *Orlando*, 1928

学习目标

- 认识到能够使用地址作为数据值的重要性。
- 能够使用指针运算符&和*。
- 能够通过引用调用在函数及其调用程序之间共享数据。
- 认识指针和数组间的关系。
- 详细理解运算符++和--的使用方法。
- 能在程序运行时应用动态分配预留新的存储空间。

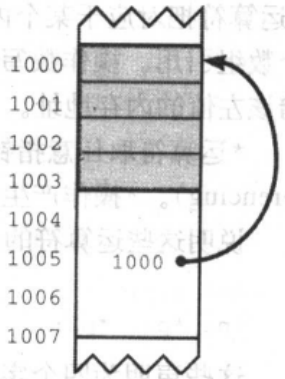
本章介绍**指针**（pointer）的概念，指针是一个数据项，它的值是其他值在内存中的地址。许多高级程序设计语言很少使用指针，因为这些语言提供了其他的机制避免使用指针。C语言的设计意图是让程序员尽可能多地访问由硬件本身提供的功能，因而指针的使用非常普遍。所以，不理解指针的工作原理就不能理解C程序。此外，如果想成为出色的C语言程序员，必须更深入地学习，学习如何在应用中更加有效地使用指针。

在C语言中，指针有多种用途，下面列出的是一些最重要的用途：

- *指针允许你以更简洁的方式引用大的数据结构*　程序中的数据结构可以任意大，但无论如何增长，数据结构总是位于计算机的内存中，因此必然会有地址。利用指针，就可以使用地址作为一个完整值的速记符号，因为一个内存地址在内部表示为一个整数。当数据结构本身很大时，这种策略能节约大量内存空间。本书在以后还会反复提到这一原则，特别是在第14章和第16章中。
- *指针使程序的不同部分能共享数据*　如果将某一个数据值的地址从一个函数传递到另一个函数，这两个函数就能使用同一数据。指针的这一应用将在本章的13.3节进一步解释。
- *利用指针，能在程序执行过程中预留新的内存空间*　到现在为止，在程序中能使用的内存就是通过显式声明分配给变量的内存。在许多应用中，如果程序能在运行时获得新的内存空间，并让指针指向这一内存则更为方便。这种策略将在本章的13.5节继续讨论。
- *指针可用来记录数据项之间的关系*　在高级程序设计应用中，指针被广泛用于构造单个数据值之间的联系。比如，程序员通常在第一个数据的内部表示中包含指向下一个数据项的指针，来说明这两个数据项之间有概念上的顺序关系。虽然对指针应用的完整讨论超出了本书的范围，但此内容在第17章中还会进行简要介绍。

了解了大多数程序设计的概念以后，掌握指针的使用要求你同时从归约和整体两方面考

程序员常在内存图上画上箭头，如右图所示。当然，在计算机内部是没有箭头的。地址1004里的字只是包含了与数值1000对应的位串。同样的位串作为整数使用还是作为地址使用取决于变量在程序中是怎样声明的。如果将变量声明为指针，就可以把存放在位置1004的值1000理解成内存中变量i的地址，并使用指针检索或操作i的值。



13.2 C语言的指针操作

C语言具有操作指针的能力，就如具有操作整数的能力一样。和任何其他数据类型一样，你可以存储指针变量中的地址值，或是将地址值作为参数传递给函数。但在此之前，考虑一些C语言中控制指针使用的细节是很重要的。

13.2.1 在C语言中声明指针变量

和C语言中其他变量一样，使用指针变量前必须先对它们进行声明。为了声明一个指针变量，需要使用下面框中的声明句法。比如，下面的声明

```
int *p;
```

将变量p声明为概念上指向整型的类型。类似的，声明

```
char *cptr;
```

表示声明变量cptr为指向字符型的类型，虽然这两种类型在计算机内部都是以地址形式表示的，但指向整型的指针和指向字符型的指针在C语言中还是有区别的。要使用该地址中的数据，编译程序必须知道如何解释地址里的数据，因此就要求显式地说明指针所指向的数据的类型。指针指向的值的类型称为指针的**基本类型**（base type）。所以，指向整型的指针的基本类型为int。

语法：指针声明

base-type ******pointer-variable*;

其中：

*base-type*是指针指向的值的类型。

*pointer-variable*就是正在被声明的变量。

必须注意，表示变量为指针的星号在语法上属于变量名，不属于基本类型。如果使用同一个声明语句来声明两个同类型的指针的话，你必须给每个变量都加上星号标志，如：

```
int *p1, *p2;
```

而声明

```
int *p1, p2;
```

则表示声明p1为指向整型的指针，而p2是整型变量。

常见错误 新程序员有时会忘记在声明指针时在每个变量前都必须加上*。比如，声明

```
char *cp,c;
```

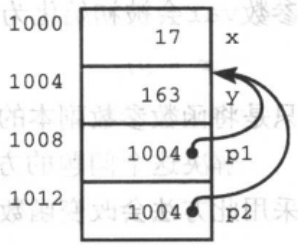
把cp声明成指向char的指针类型，而c是char类型的变量。

13.2.2 基本的指针操作

C语言定义了两种操作指针值的运算符：

- & 取地址。

命令计算机取出变量p2的值并将该值复制给变量p1。p2的值是指针值1004。如果你将这个值赋值给p1，那么p1和p2就会同时指向变量y，如右图所示。就发生在机器内部的操作来说，复制一个指针和复制一个整数是完全一样的。指针的值被原封不动地复制到目的地。从图的概念角度来说，复制指针的效果就是将目的地指针处的箭头替换成与原指针指向同一位置的箭头。所以，赋值语句




```
p1 = p2;
```

的效果就是改变了从p1出发的箭头，使其和从p2出发的箭头指向同一内存。 460

能够区分指针赋值和值赋值是很重要的。形如

```
p1 = p2;
```

的指针赋值使p1和p2指向同一位置。而表示为语句

```
*p1 = *p2;
```

的值赋值是把以p2为地址的内存位置里的值复制到以p1为地址的内存位置里去。

13.2.3 特殊指针NULL

在指针的应用中，特别是在超出本书的范围的应用中，能在一个指针变量中存储一个特殊值表示变量实际上并不指向任何有效数据是很有用的。为了达到这个目的[⊖]，C语言定义了一个特殊的常量NULL。常量NULL可以赋给任何指针变量，在机器内部表示为地址值0。

如果一个指针变量的值为NULL，很重要的一点就是，不要用*运算符来间接引用此变量。NULL值的目的是表示指针不指向有效数据，所以想要找出和NULL指针有关的数据是没有意义的。遗憾的是，大多数编译器没有查找这一错误的程序。如果你间接引用NULL，通常计算机会寻找存放在内存地址0中的值。如果你正巧使用NULL指针赋值来改变那个值的话，程序会崩溃，也不会有任何关于问题如何产生的线索。未初始化的指针变量也会发生同样的问题。

NULL指针的各种使用会在介绍和它们相关的应用时介绍。现在要记住的是，有这样一个常量存在。

常见错误 小心不要间接引用那些没有初始化或值为NULL的指针，这样做会引用不属于本程序的内存空间，很可能会使程序崩溃。

13.3 通过引用传递参数

为了理解指针变量在实际中是如何使用的，让我们看一下C语言中指针的最常见的应用之一，即传递指针给函数，允许该函数操作其调用程序的数据。在C语言中，把一个简单变量从一个函数传递到另一个函数时，该函数就得到了一个调用值的拷贝。在函数的语句中给参数赋一个新值会改变参数在这个函数中的局部副本，但不会影响调用参数。例如，如果要实现 461
一个用以下方法将某个变量初始化为0的函数：

```
void SetToZero(int var)
{
    var = 0;
}
```

（此程序是有错的，事实上无效）

该函数不起任何作用。如果调用

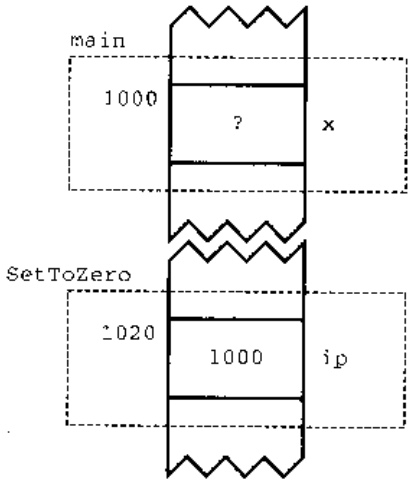
⊖ 常量NULL事实上是在stdlib.h头文件中定义的，当程序包含stdio.h头文件时，就会自动包含这个头文件。因为本书中所有程序都包含stdio.h，所以你可以把NULL当成是C语言内置的常量。

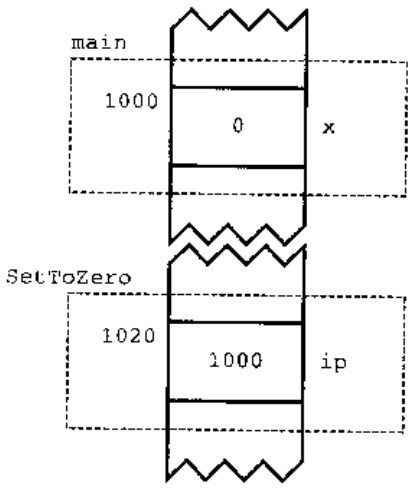
这个帧同样也处于内存的某处，因此内存应该如下面的左图所示[⊖]。

语句

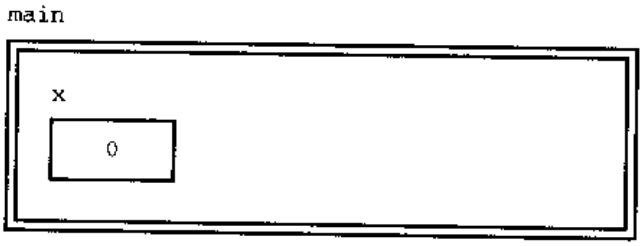
```
*ip = 0;
```

能把地址为ip的整型字的值置为0，如下面的右图所示：





当SetToZero函数返回时，地址1000中值的改变还是有效的，所以main函数的帧现在如下图所示。指针作为参数使函数能够改变其调用函数的帧中的值。在C语言中，你必须通过声明参数值为指针类型明确说明想要改变值，然后将地址作为参数传递。在很多其他语言中，这种机制是语言定义的一部分，称为**引用调用**（call by reference）。虽然C语言的方法并不那么方便，并且需更长的时间来学习，但它确实有自己的优势。其最重要的优势在于函数调用自身的句法指出了参数变量值在函数执行的过程中是否能被改变。例如，在完全不了解函数Mystery的情况下，你却可以知道变量x的值在进行如下调用后没有发生改变：




```
Mystery(x);
```

如果Mystery要改变x的值的话，函数必须重新定义为有一个指针参数，则调用就变为

463
|
464

```
Mystery(&x);
```

应用这一规则使得我们容易预测某个函数调用的结果。

13.3.1 设计函数SwapInteger

为了说明引用调用是怎样影响程序设计的，让我们回到图12-5表示的选择排序算法。SortIntegerArray过程的实现如下：

⊖ 实际上哪个地址会分配给这些帧是没有办法预测的，甚至连这两个帧的相对方位也是无法预测的。在大多数现代的计算机，一般SetToZero帧的内存地址比main帧的内存地址小，但理解内存结构并不依赖于任何这样的假定。

```
/*
 * File: hours.c
 * -------------
 * This program converts a time given in minutes into
 * separate values representing hours and minutes.  The
 * program is written as an illustration of C's mechanism
 * for simulating call by reference.
 */

#include <stdio.h>
#include "genlib.h"
#include "simpio.h"

/* Constants */

#define MinutesPerHour 60

/* Function prototypes */

static void ConvertTimeToHM(int time, int *pHours, int *pMinutes);

/* Test program */

main()
{
    int time, hours, minutes;

    printf("Test program to convert time values\n");
    printf("Enter a time duration in minutes: ");
    time = GetInteger();
    ConvertTimeToHM(time, &hours, &minutes);
    printf("HH:MM format: %d:%02d\n", hours, minutes);
}

/*
 * Function: ConvertTimeToHM
 * Usage: ConvertTimeToHM(time, &hours, &minutes);
 * -----------------------------------------------
 * This function converts a time value given in minutes into
 * an integral number of hours and the remaining number of minutes.
 * Note that the last two arguments must be passed using their
 * addresses so that the function can correctly set those values.
 */

static void ConvertTimeToHM(int time, int *pHours, int *pMinutes)
{
    *pHours = time / MinutesPerHour;
    *pMinutes = time % MinutesPerHour;
}
```

图13-1 hours.c

在这个实现中，这些值被表示为指向实际变量的指针。所以，为了计算小时数，合适的赋值语句为：

```
*pHours = time/MinutesPerHour;
```

你要用*运算符来保证该结果存储在pHours指向的变量中。

是一个左值，所以书写

```
&list[i]
```

是合法的，它表示list中第i个元素的地址。

由于list中第i个元素的地址取决于变量i的值，所以C语言编译器在编译程序时是无法计算这一地址的。为了确定这一地址，编译器产生一段指令，取数组的基地址，加上适当的偏移量，偏移量是将i的值乘以每个数组元素的字节数而得到的。所以，计算list[i]的地址可以由公式

$$1000 + i \times 8$$

得到。例如，如果i为2，地址计算的结果即为1016，与图中显示的list[2]的地址是一致的。因为计算任何一个数组元素地址的过程是自动的，所以在写程序时无需考虑计算的细节问题。

13.4.1 指针运算

在第2章介绍运算符+和-时，你很可能还未认真地考虑过它们。毕竟，从小学开始就开始使用这两个运算符了。当+和-的操作数为数字时，结果由简单的数学运算决定。在C语言中，469
也可以将这些运算符应用于指针，在某些方面，其结果与数学运算相似，但在另外一些方面又有所不同。对指针值应用加减的过程称为**指针运算**（pointer arithmetic）。

下面的规则对理解指针运算工作机制是很有必要的。

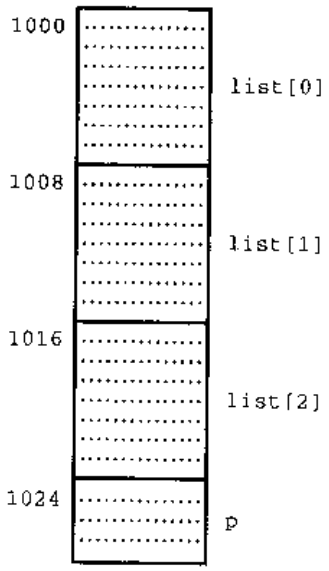
> **指针运算规则** 如果指针p指向数组arr的第一个元素且k为整型数，以下的等式总是成立的：p+k定义为&arr[k]。

换句话说，如果你在指针值上加上一个整数k，则结果就是下标为k的元素的地址。

为了解释指针运算规则，假设有一个函数包含如下所示的变量声明：

```
double list[3];
double *p;
```

在函数帧内给每个变量都分配了空间。对数组变量list，编译器为它的三个元素都分配了足够的空间以便存放一个double型的数据。对指针p，编译器为其分配了足够空间来存放某个double类型的左值的地址。如果帧从地址1000开始，内存分配如右图所示。



由于还未给这些变量赋值，所以无法确定其初始内容。但是可以给它们赋值。例如，可以使用以下的赋值语句将任意值存储到每个数组元素中去：

```
list[0] = 1.0;
list[1] = 1.1;
list[2] = 1.2;
```

要初始化指针变量p，使其指向数组的起始地址，可以执行以下赋值语句：

```
p = &list[0];
```

13.4.2 运算符++和--的新作用

在深入探索指针的神秘之处之前，有必要重新看一下自增和自减运算符，即在第3章中介绍的++和--。到目前为止，本书中的程序仅将这些运算符用于操作数，对它们所操作的左值加1或者减1。

但在实际情况中，这些运算符可以比前面例子中出现的情况更加复杂。首先，这两个运算符都可以用两种方式写，我们熟悉的一种写法是将运算符写在操作数后，即

```
x++
```

或者将运算符写在操作数前，即

```
++x
```

第一种形式，即运算符在操作数之后，称为**后缀**（postfix）式，而第二种形式称为**前缀**（prefix）式。

如果所要做的只是执行单独++运算，就像把它作为一个单独的语句，或作为for循环中的自增运算符那样，前缀运算符和后缀运算符的效果是完全一样的。只有在把这两个运算符作为更长的表达式的一部分时，区别才显现出来。和所有的运算符一样，++运算符会返回一个值，但为何值取决于运算符相对于操作数的位置。这两种情况如下：

- x++ 先计算x的值再将它加1。返回给外部表达式的值是加1前的原始值。
- ++x 先将x的值加1再使用新值作为整个++操作所得的值。

--运算符也类似，但为减1而非加1。

472
~
473

假如执行以下程序：

```
main()
{
    int x, y;

    x = 5;
    y = ++x;
    printf("x = %d, y = %d\n", x, y);
}
```

结果为

```
x = 6, y = 6
```

如果程序写为

```
main()
{
    int x, y;

    x = 5;
    y = x++;
    printf("x = %d, y = %d\n", x, y);
}
```

则最后结果为

```
x = 6, y = 5
```

语言中最常用的一种结构，即表达式：

```
*p++
```

要确定该表达式意义，首先需要解决的问题就是操作的顺序是什么。*运算符和++运算符会争夺操作数p。按照语言的优先级规则和结合性规则，该表达式可能会等同于 475

```
(*p)++
```

或者是

```
*(p++)
```

当这种情况发生时，C语言中的一元运算符按照从右至左的顺序进行运算。所以，++运算优先于*运算，第二种解释是正确的。但是你还是需要确定表达式到底是什么意思。

如以前所见，后缀式++运算符让p值先自增，随后再返回自增操作前的p值。由于p是一个指针，你必须根据指针运算定义自增操作，所以，在计算p+1时，你知道结果值应该指向数组中的下一个元素。如果p原来指向arr[0]，那么自增操作就会使其指向arr[1]。于是表达式

```
*p++
```

表示下面的含义：

> 间接引用指针p，使其当前指向的对象作为左值返回。其负面影响是，p值进行了自增，所以如果原左值是数组的一个元素的话，新的p值指向数组中的下一个元素。

但是为了理解该运算符的有用之处，必须更细致地考虑指针和数组之间的关系。

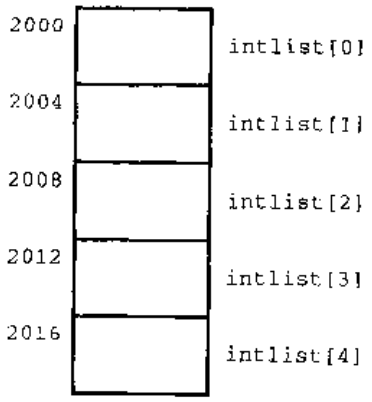
13.4.4 指针和数组的关系

C语言最不寻常的特征中，最有趣的是数组名作为指向该数组第一个元素的指针的同义词。这个概念很容易用例子解释。

声明

```
int intlist[5];
```

为一个含有5个整型数的数组分配了空间，如右图所示，分配的空间在计算机内存的某处。图中左边的地址是任意的。在这个图中，起始地址是2000，但也可以是其他的任意地址。



intlist代表一个数组，但也可以直接用作指针值。当它作为指针使用时，intlist定义为数组中第一个元素的地址。对于任意数组arr来说，下述等式在C语言中是永远成立的：

```
arr 等同于 &arr[0]
```

给定任何数组名，可以将它的地址直接赋给任何指针变量。当一个数组从一个函数传递到另一个函数时通常可以用到这个等式。典型地，被调用函数利用第12章中Sort原型所示的句法来声明这个数组：

```
void SortIntegerArray(int array[], int n);
```

如果你想要用上面定义的intlist数组调用Sort，那么传递给Sort的形式参数array的值将会是intlist中第一个元素的地址。如果将原型写成

```
void SortIntegerArray(int *array, int n);
```

节的内存，可调用

```
malloc(10)
```

结果会返回一个指针，指向一个10字节大小的内存块。为了使用新分配的空间，必须将malloc的结果存放在一个指针变量内，以后就可以像使用数组一样使用该指针变量了。

13.5.1 Void*类型

但在给出应用malloc函数的例子之前，先让我们讨论一个很可能会十分棘手的问题。在C语言中，指针是具有类型的。如果通过

```
int *ip;
```

478

声明一个变量ip，该变量在概念上是一个指向整型的指针。而声明

```
char *cp;
```

则引入了一个指向字符的指针变量cp。在ANSI C中，这些类型是不同的，如果你想要把其他类型赋给这些指针变量的话，编译器会发出警告信息。

指针有类型的概念引发了一个很有趣的问题，即malloc函数返回的是什么类型的指针呢？malloc函数用于为调用函数需要的任何类型的值分配新的内存空间，所以必须返回一个未确定类型的“通用”指针。在C语言中，通用指针类型是一个指向空类型void的指针，void也用于指示无返回值或参数列表为空的函数。如果声明一个指向void类型的指针，如

```
void *vp;
```

你可以将任何类型的指针值存入该变量，但不允许用*运算符间接引用vp。编译器不知道vp的基本类型是什么，所以没有办法谈论vp指向的值。

虽然如此，void*类型仍是有用的，因为它允许函数，特别是malloc函数，返回以后由调用函数创建实际类型的通用指针。malloc函数返回类型为void*的指针值，表明其原型为[⊖]：

```
void *malloc(int nBytes);
```

注意，这个函数原型的结果类型与指针变量的声明方法非常相似。*表示结果是和函数名而非基本类型相联系的指针，尽管在概念上，结果是与基本类型而非函数名联系的。除非习惯C语言中返回指针值的函数句法，否则是很容易混淆的。

ANSI C能在指向void的指针类型和指向基本类型的指针类型间自动进行转换。例如，如果声明字符指针cp为：

```
char *cp;
```

就可以用语句

```
cp = malloc(10);
```

将malloc的结果直接赋给cp。许多程序员在赋值之前，会使用下列强制类型转换将malloc返回的结果转换成指向字符的指针。这样做的一部分原因是因为历史因素，另一部分原因是这样做能使指针类型间的转换更清楚。

479

```
cp = (char*)malloc(10);
```

不管有没有明确使用强制类型转换，这条语句均可分配10个字节的新内存空间，并将第一个

⊖ malloc的参数类型实际上是在头文件stddef.h中定义的size_t类型。在一些机器上，int类型的大小可能不足以表示内存块的大小。不论在哪种情况下，参数在概念上是整数。

因为动态分配往往会频繁用于多种程序，因此错误检查尽管重要，却也是极其单调乏味的。而且常常因为出错消息使代码变得混乱不堪，搞乱算法结构。实际上，当空间用完时，就无计可施了，程序显示出错消息并且停止运行通常是唯一可行的方法。所以，调用malloc并将其嵌入一个包括内存不够测试的新抽象层中是很有用的。

genlib.h接口输出函数GetBlock，该函数结合了malloc的作用，并对NULL结果进行测试。如果GetBlock函数检测到内存不够的情况，就会调用Error函数。所以，如果以上两行代码用

```
arr - GetBlock(10 * sizeof(int));
```

代替的话，GetBlock函数的执行就结合了分配内存和检测错误的操作。因为GetBlock使内存分配的操作更清晰，程序更易懂，因此本书多采用GetBlock而不是malloc。但在概念上，这两个函数的功能是相同的。

为了进一步简化分配动态数组的过程，genlib.h库还定义了函数NewArray。NewArray取元素数和其基本类型后返回一个指针，指向特定大小的动态数组。所以，要分配一个含50个字符串的动态数组，应调用NewArray(50, string)。

13.5.4 释放内存

保证不会发生内存不够的一种方法是一旦使用完已分配的空间就立刻释放它。标准ANSI库提供了函数free，用于归还以前由malloc分配出去的堆内存。例如，如果你能肯定已经不再使用分配给arr的内存，就可以通过调用

```
free(arr);
```

481

释放该空间。但由于本书应用GetBlock执行动态分配，所以为分配和释放建立一个包括兼容的命名机制的统一抽象体系尤为重要。为此，genlib.h接口也包括一个FreeBlock函数，其操作方法和free完全相同。

但事实证明，知道何时释放一块内存并不那么容易。根本问题在于分配和释放内存的操作分别属于接口两边的实现及客户。实现知道何时该分配内存，返回指针给客户，但它并不知道何时客户结束使用已分配的对象，所以释放内存是客户的责任，虽然客户可能并不十分了解对象的结构。

对现在的大多数计算机的内存来说，你可以随意分配所需内存而不需要考虑释放内存的问题。这种策略几乎对所有运行时间不长的程序还是有效的。内存有限的问题只有在设计一个需要运行很长时间的应用，比如所有其他系统所依靠的操作系统时，才变得有意义。在这些应用中，不需要内存时释放它们是很重要的。

某些语言支持那些能主动检查正在使用的内存，并释放不再使用的内存的动态分配系统。此策略称为**碎片收集**（garbage collection）。C语言中也有碎片收集分配器，而且它未来的应用也许会更广泛。如果是这样的话，即使在长时间运行的程序中，也可以忽略释放内存的问题，因为你可以依靠碎片收集器自动执行释放内存的操作。

在大部分情况下，本书假定无论何时都可以给你的应用分配内存，这样大大简化了程序，并能使你专注于算法细节。

小结

本章介绍了指针的概念。指针是计算机内存中一个数据值的地址。指针通常是很有用的，

```
double d1;
double d2;
double *dp1;
double *dp2;
```

在图中，跟踪这些语句的操作：

```
dp1 = &d1;
dp2 = &d2;
*dp1 = 3.14159;
d2 = 2.71828;
dp1 = dp2;
*dp1 -= d2;
```

7. 判断题：对于任意变量x，表达式*&x都为x的同义词。

8. 判断题：对于任意变量x，表达式&*x都为x的同义词。

9. 常量NULL的内部表示是什么？

10. 间接引用NULL指针时会发生什么？

11. 引用调用的含义是什么？

12. 在不了解函数ApplySpecialOperation的作用的前提下，确定整型变量x和y中哪一个可能受到调用

```
ApplySpecialOperation(&x,y)
```

的影响。

13. 在什么情况下会使用引用调用？

14. 假设intArray声明为

```
int intArray[10];
```

且j是一个整型变量，描述计算机确定下列表达式的值时将采取的步骤。

```
&intArray[j+3];
```

15. 如果arr被声明为一个数组，描述以下两个表达式之间的区别：

```
arr[2]
arr+2
```

484

16. 假设double类型的变量在你使用的计算机系统中占8个字节。如果数组doubleArray的基地址为1000，那么doubleArray+5的地址值是什么？

17. 假如整型变量x和y的值分别为2和5，试确定以下几种情况下表达式的值和执行表达式后对x和y的值的影响。（在回答这个问题时，每一表达式都单独处理，而不是将其作为连续的表达式。换句话说，每次计算前x和y的值都是2和5。）

a. x++

b. -x

c. x++ + ++y

d. y + = x--

18. 给出一个包含++运算符的表达式，不同机器解释这个表达式会产生不同的结果。

19. 判断题：如果p为指针变量，那么表达式p++在p的内部表示中增加了1。

20. 表达式

```
*p++
```

```
Enter the elements of the array, one per line.
Use -1 to signal the end of the list.
? 67↵
? 78↵
? 75↵
? 70↵
? 71↵
? 80↵
? 69↵
? 86↵
? 65↵
? 54↵
? 76↵
? 78↵
? 70↵
? 68↵
? 77↵
? -1↵
The range of values is 54-86
```

4. 重写图12-5表示的选择排序的实现，将所有的数组引用改成指针引用。完成后，方括号字符“[”和“]”不应该出现在源文件中。但即便如此，SortIntegerArray函数的作用应和以前所述相同。

5. 写一个函数IndexArray(n)，返回指向动态分配的含n个元素的整型数组的指针，每个元素均初始化为自身的下标。例如，假设ip声明成：

```
int *ip;
```

那么语句

```
ip = IndexArray(10);
```

应该产生如下内存配置：



6. 写一个函数Tabluate(array, n)，使其取一个整型数组和它的有效大小作为参数，画出一张显示每个元素出现次数的表格。例如，假设输入数组为

91	93	98	92	92	95	93	92	91	95	99	92	98

则程序应能产生如下表格：

```
91: 2
92: 4
93: 2
95: 2
98: 2
99: 1
```

第14章 再论字符串

. . . Untune that string
And, hark! what discord follows;

— Shakespeare, *Troilus and Cressida*

学习目标

- 理解字符串数据的基本表示方法。
- 了解字符串变量声明为数组和声明为指针之间的区别。
- 能够使用ANSI字符串库中的函数。
- 学会如何用string.h函数实现strlib.h接口。

490 ~ 491

从第2章我们就开始使用字符串了，从第9章开始已经能够执行高级的字符串操作。但是到目前为止，本书中的程序只将string作为一种抽象的类型。作为一种抽象的类型，string类型的值是一个完整的、不可分割的实体。可以将字符串传递给函数，也可以将字符串作为结果返回，还可以将一个字符串的值赋给另一个字符串。如果想要从字符串中选出一个字符或是进行更加复杂的字符串操作，就必须调用strlib.h接口导出的某个函数。

和其他接口一样，strlib.h的主要目的之一是对客户隐藏那些不必知道的细节。但是，由于字符串操作是许多程序设计应用的重要方面，因此必须学会在较低的细节层次上使用字符串。为了完全弄清字符串的工作原理，有必要超越抽象的层次，思考其根本的表示方法。

14.1 string类型的概念表示

在C语言中，为了更有效地使用字符串，必须能从三个角度考虑它：

1) 作为字符数组。

2) 作为指向单个字符的指针。

3) 作为具有整体的概念完整性的完整实体。

这三个观点不是矛盾的，而是互补的，每一个观点都描述了一个现实的角度。对于一个存储在计算机中的字符串，可以选择这三种观点中的任何一种来考虑。因为只是从不同的角度来考虑字符串，因此字符串本身并不发生变化，唯一改变的只是在你的概念中如何表示字符串。

14.1.1 字符串作为数组

在计算机内部，字符串被表示为字符数组。只要将一个字符串存储到内存，这个字符串中的字符就都被分配到连续的字节中。例如，当编译器在一个程序中看到字符串常量"Hello"时，就会把组成这个字符串的字符存入内存中连续的字节空间内。

但是，存放字符本身并不足以表示所有与字符串有关的重要信息。操作字符串值的程序必须具有某种确定字符串结束的方法。如果编译器要将字符串常量"there"中的字符直接存储在字符串常量"Hello"的字符之后，内存看起来如下图所示：

492

```
    ...操作str[i]的循环体...
}
```

在每个循环周期中，选择表达式str[i]指的都是字符串中的第i个字符。由于循环的目的是处理每一个字符，循环会一直持续到str[i]选择了标志字符串结束的空字符。所以这个for循环在功能方面等价于第9章中介绍的如下习语：

```
for (i = 0; i < StringLength(str); i++) {
    ...操作IthChar (str, i) 的循环体...
}
```

利用数组选择形式比使用strlib.h接口中的函数有效得多。首先，数组选择避免了调用IthChar函数所需付出的代价。而且，用

```
i < StringLength(str)
```

作为for循环控制语句中的测试条件意味着，即使字符串长度通常是不变的，程序也必须在每个循环周期中都重新计算一遍字符串长度。通过检查结束标记，程序可以避免这些冗余的计算。 493 到 494

为了对这两种方法进行比较，可以不采用任何strlib.h接口中的函数来重写图10-5中的Pig Latin一例中介绍的FindFirstVowel函数。图14-1中是第10章中原来的实现。

如果以数组符号写这个函数的话，它的实现如图14-2所示。

这两个程序在算法上是完全相同的，只在代码的细节上有些微区别，主要是检测字符串结束和选择每个字符的方法有所不同。

```
int FindFirstVowel(string word)
{
    int i;

    for (i = 0; i < StringLength(word); i++) {
        if (IsVowel(IthChar(word, i))) return (i);
    }
    return (-1);
}
```

图14-1 用抽象字符串实现FindFirstVowel

```
int FindFirstVowel(char word[])
{
    int i;

    for (i = 0; word[i] != '\0'; i++) {
        if (IsVowel(word[i])) return (i);
    }
    return (-1);
}
```

图14-2 基于数组的FindFirstVowel的实现

14.1.2 字符串作为指针

和C语言中的任何数组一样，字符数组也可以被解释为指向其首元素的指针。给定一个指向字符串值的指针，就可以用指针运算引用该字符串中的各个字符。因此，FindFirstVowel也可以写成如图14-3所示的代码。

符。由于该函数的实现使用数组符号，所以它的形参被声明为如下的字符数组：

```
int FindFirstVowel(char word[])
```

图14-3中的实现将参数作为指针，如函数头一行的声明所示：

```
int FindFirstVowel(char *word)
```

就C语言编译器而言，这三个函数头中的参数声明是完全一样的，可以互换使用。选择某个表示方法的价值在于这种表示方法传达给读者的额外信息。通常，声明形参应该和这些参数的使用风格一致。如果把字符串作为一个字符数组考虑，并用方括号选择各个字符的话，应该将该字符串声明成一个数组。如果把字符串作为一个指针考虑，并用*间接引用这个指针的话，应该将该字符串声明成一个指针。如果把字符串作为一个完整实体考虑，并且这个完整实体比其各个组成部分更重要的话，最好的方法是将该字符串声明成string类型。

14.1.5 字符串变量

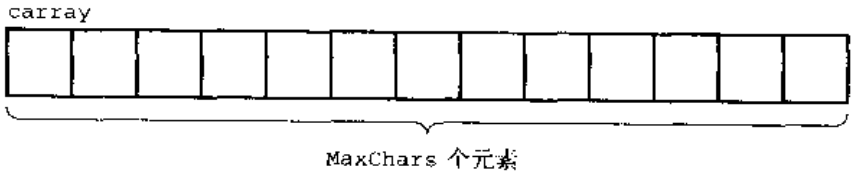
尽管在C语言中数组和指针经常可以互换使用，我们必须认识到除非变量是上一节描述的形参，否则将这个变量声明成数组和将其声明成指针是截然不同的。对于所有其他的变量类型来说，所选择的声明方式决定了内存的分配方式。

如果用以下代码来声明一个数组：

```
char carray[MaxChars];
```

497

那么为carray分配的空间就在当前函数帧内。如下图所示，你有MaxChars个字节可用于存放字符：



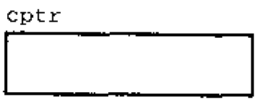
由于内存是显式分配的，因此可以直接给carray的元素赋值，也可以将carray数组传递给其他函数，这样就能自由操作分配到的内存了。

如果你换一种方式，将一个变量cptr声明为

```
char *cptr;
```

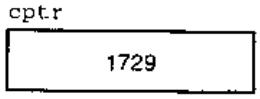
或者（等同地）声明为

```
string cptr;
```



则完全没有为字符分配任何内存。变量cptr只是一个指针变量（如右上图所示）。

因为这一变量没有显式地初始化，所以它的内容是不确定的。它可能包含0、42或任何以前此位置被使用时所留下的随机值。假设cptr中包含的正好是1729，如右下图所示。如果改变cptr[0]的值，将会改变地址1729里所存储的值。但是，地址1729可能被程序中的某个变量或某一部分代码使用。这样进行赋值的话会很容易破坏关键数据并使程序崩溃。



常见错误 声明一个指针变量时，必须确定你已对其进行了初始化，这样才能让它指向内存的真实位置。未成功初始化指针是造成错误的常见原因。这些错误是很难发现的，因为编译器不能捕捉到它们。最好的方法是完全地避免它们。

使用cptr之前，你必须提供必要的字符内存，方法是给其赋一个和可用内存区域的地址

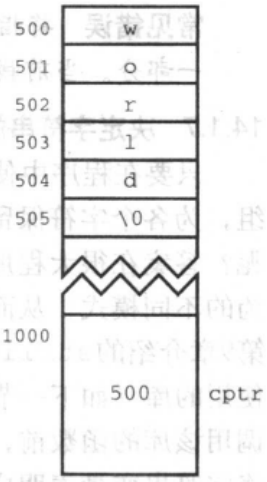
来分配字符串内存的话，形如

```
carray = "hello";          (这条语句是非法的)
```

的赋值语句是非法的。因为变量carray不是左值，你无法给它赋值。

将字符串声明成指针就没有这种限制了。例如，声明char *cptr;后，cptr="world";是完全合法的。

常见错误 在C语言中，数组名不是左值。所以把值赋给任何数组名都是非法的，你必须将值赋给单个数组元素。特别是，如果你把一个字符串声明为字符数组，就不能直接将字符串值赋给该数组。



编译器将"world"的字符存在内存某处，赋值语句将字符串第一个字符的地址存储在cptr中。例如，如果编译器将"world"的字符存放在从500开始的内存位置，给cptr赋值的效果可以用右图表示。变量cptr只是经过更新指向了一个新的位置。

调用函数时也有类似的限制。尽管函数可以将一个数组作为参数，但在C语言中函数返回一个数组是非法的。然而，函数可以返回指针，从而达到同样的效果。如果曾经在函数里用动态分配为某个数组分配内存，则可以将指向该内存的指针作为函数的值返回，将结果赋给一个指针变量，接着就能像使用数组一样使用这个指针变量。

举例来说，strlib库中的函数CharToString可以用如下方式实现：

```
string CharToString(char ch)
{
   char *cptr;

   cptr = GetBlock(2);
   cptr[0] = ch;
   cptr[1] = '\0';
   return (cptr);
}
```

第一条语句从堆中分配了两个字节的内存：一个字节给字符，另一个字节给每个字符串末尾都必须出现的空字符。接下来两条语句把正确的值赋给这两个字符。最后一条语句将一个指向新分配内存的指针作为函数值返回。接着，客户写形如

```
star = CharToString('*');
```

的语句就是合法的，该语句假设star被声明为字符指针，或者（等同地）被声明成抽象类型string。

认识到这个方法只适用于动态分配数组空间是很重要的。如果想要返回一个指向在函数中声明的数组的指针，程序将会失败。问题在于分配给声明成局部变量的数组的内存在函数返回时就自动释放了。

要理解这个问题，想象你用以下这个错误的方法实现CharToString：

```
string CharToString(char ch)
{
```

任务，但学会怎样使用string.h接口不仅仅是记住一组新的函数名。string.h接口是建立在一个不同的字符串操作的模型上，要求你从一个新的角度看待字符串。

两个字符串库的最大区别在于怎样为字符串中的字符分配内存空间。在strlib.h接口中，函数本身就会分配必要的内存空间。作为客户，你无需为分配的细节担忧，这样就使得strlib.h接口更容易使用。string.h接口中的函数将分配内存的重任转移到了客户身上。当一个新字符串需要内存时，客户要负责提供内存。为此，客户通常会声明一个足以保存结果的字符数组，并将它作为参数传递给库函数。然后该实现会把新字符串值写入调用函数提供的内存空间中。

表14-1 string.h输出的常用函数

函数调用	操作
strcpy(dst, src)	将字符从src复制到dst
strncpy(dst, src, n)	最多将n个字符从src复制到dst
strcat(dst, src)	将src追加字符至dst后面
strncat(dst, src, n)	最多将n个字符从src追加至dst后面
strlen(s)	返回s的长度
strcmp(s1, s2)	返回一个整数，表示比较的结果
strncmp(s1, s2, n)	类似于strcmp，但最多比较n个字符
strchr(s, ch)	返回一个指针，指向s中的第一个ch的实例（或者返回NULL）
strrchr(s, ch)	返回一个指针，指向s中的最后一个ch的实例（或者返回NULL）
strstr(s1, s2)	返回一个指针，指向s1中的第一个s2的实例（或者返回NULL）

502 ~ 503

14.2.1 strcpy函数

string.h接口中的strcpy函数为库中函数如何给调用函数返回一个新的字符串值提供了最好的方法。strcpy函数将一个源（source）字符串中的字符复制到另一个目标（destination）字符串中。为了和赋值运算符相一致，复制操作是从右向左进行的，strcpy将目标参数作为第一个参数，因此就有了以下原型[⊖]：

```
void strcpy(string dst, string src);
```

如果你想要操作一个字符串，同时又保留它的原值的话，strcpy函数是很有用的。例如，假设字符串变量line含有一行从用户处读入的文本，程序要求你对line中的字符做一些修改。直接用数组选择来更改line中的元素会破坏line原来的内容，而且你以后还有可能会用到这些内容。所以，为了避免改变line，可以将其中的各个字符复制到另外一个数组再进行操作。

保存数据中间副本的数组称为**缓冲区**（buffer）。在这种情况下，我们需要的是一个字符缓冲区，以便装入line的数据副本。当使用ANSI字符串库时，通常的方法是明确声明一个字符数组作为缓冲区，大小足以存放应用中可能遇到的最大字符串。所以，如果常量MaxLine含有最长输入串的长度，你可以这样声明缓冲区：

```
char buffer[MaxLine+1];
```

此处的+1用来为长度正好是MaxLine的字符串末的空字符留出空间。

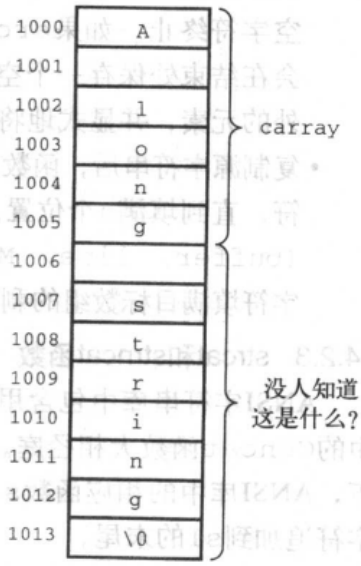
重要的是，要牢记不能采用如下赋值语句将line的字符复制到buffer中：

⊖ 和库中的定义一样，strcpy函数确实返回了目标数组的地址，尽管大多数客户会忽略这个值。为了表达的简洁性，本文将strcpy函数定义成一个无结果的过程。

```
char carray[6];

strcpy(carray, "A long string");
```

执行这一程序时会将"A long string"的前六个字符正确复制到carray中，但是，程序 505
又继续把剩余的字符复制到内存中carray之后的字节里面，所以，调用strcpy之后，内存如右图所示。地址1006～1013间的字节存放的也许是其他变量或是程序本身的一部分，所以改变它们肯定会引起程序失败。写入的数据超出作为缓冲区的数组的大小是一种常见的程序设计错误，这种错误称为**缓冲区溢出**（buffer overflow）。



如果目标字符串是一个未妥善初始化的指针变量，就会发生同样严重的问题。例如，假设用语句

```
string str;
```

声明了一个字符串变量str，那么就不能用strcpy复制字符串到该变量中。如果试图写语句

```
strcpy(str, "A long string");        (此语句无效)
```

并且没有事先初始化变量str使其指向某一字符数组，"A long string"会被复制到内存中无法预知的区域。

作为程序员，你的责任是确保不会发生这些错误。如果你认为已经分配给目标字符串的空间可能不够，就必须在使用strcpy复制字符串之前检查数据的长度。所以，在用strcpy将line的内容复制到buffer之前，应该通过测试语句行确保不会发生缓冲区溢出的情况，测试如下：

```
if (strlen(line) > MaxLine) Error("Input line too long");
strcpy(buffer, line);
```

506

如果没有检测到缓冲区溢出的话会发生可怕的后果。1988年，康奈尔大学的一名研究生发布了一个程序，它像病毒一样，通过连接在因特网上的计算机间迅速传播。这个程序就是我们所知道的因特网蠕虫病毒。因特网上许多电脑的系统设备程序都不会检查机器内部缓冲区的字符内存是否被输入数据用尽，蠕虫病毒正是利用了这一点，通过输入大量数据用尽已分配内存，然后覆盖系统程序本身，并执行利于自身的命令。

14.2.2 strncpy函数

为了避免在使用strcpy函数时缓冲区溢出，ANSI字符串库包含了另一种形式的strcpy，称为strncpy。strncpy函数允许客户指定一个长度限制，其原型如下：

```
void strncpy(string dst, string src, int n);
```

和strcpy一样，strncpy将src指定的字符串中的字符复制到dst指定的字符数组中。区别在于strncpy最多从src复制n个字符，遇到空字符会提前停止。strncpy允许你指定字符串的最大长度，所以能防止写入的字符超过字符数组的大小。例如，如果通过语句

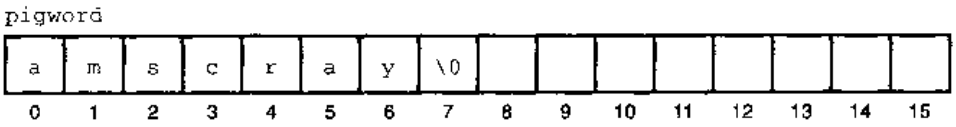
```
char buffer[MaxLine+1];
```

声明字符数组buffer，则可以用函数调用

最后一步，调用 508

```
strcat(pigword, "ay");
```

完成Pig Latin词汇，结果如下：



和使用strcpy一样，你必须小心使用strcat函数，以避免连接过多字符，从而超出缓冲区所能提供的大小。字符串库包含的函数strncat有一定的作用，但是决不如你想象的那么多。调用strncat(s1, s2, n)最多能从s2复制n个字符到s1末尾，但为了确定给n赋予何值，还需检查s1的当前长度，这样才能知道缓冲区还剩余多少空间。

14.2.4 strlen、strcmp和strncmp函数

string.h输出的strlen和strcmp函数与第9章介绍的StringLength和StringCompare完全一样。使用ANSI库时，可以像使用StringLength和StringCompare一样使用strlen和strcmp。

但string.h接口没有包含与StringEqual直接对应的函数。要测试两个字符串是否相等，C语言程序员只需调用strcmp并查看结果是否为0。当程序能明确地将结果和0作比较时，这个方法不会引起混淆。但如果程序员依赖C语言中把整数0解释成FALSE，而把其他整数解释成TRUE的话，程序就难以读懂了。例如，有时会看到如下if语句：

```
if (strcmp(s1, s2)) {
    ...语句...
}
```

当s1和s2不相同时，就会执行if语句的内容。

图12-2中给出的FindStringInArray的实现给出了紧凑编程风格的含义更为模糊的例子：

```
int FindStringInArray(string key, string array[], int n)
{
    int i;

    for (i = 0; i < n && strcmp(key, array[i]); i++);
    if (i < n) return (i);
    return (-1);
}
```

只要i比n小，且key与当前数组项不符，for循环就会一直执行下去。在两种情况下可以退出for循环：数组中的元素都已测试完毕，或者是循环在下标i处找到了匹配的元素而终止。由于for控制行已包括了所有所需的工作，所以循环体只用了一个分号。 509

string.h接口也包含另一个字符串比较函数strncmp，它有时会非常有用。调用strncmp(s1, s2, n)和调用strcmp(s1, s2)相似，只是strncmp最多只考虑两个字符串中的n个字符。如果字符串的前n个字符都相同，strncmp就返回0，就算以后的字符不同也是如此。

14.2.5 strchr、strrchr和strstr函数

string.h接口也输出一些用于搜寻字符串的函数。函数strchr(s, ch)和FindChar(ch, s, 0)的功能相同。它们主要的区别是，strchr返回一个指向匹配字符的指针，而FindChar返回该字符的下标。如果没有找到匹配字符，strchr返回NULL。函数strrchr

以便开始学习标准接口。

在这种情况下，通常的做法是让一个接口重新命名比它更低层的接口的函数。这样的函数就称为**转换函数**。提供一个传递函数而不让客户使用较低层次的对应函数的目的是为了减少客户在概念上的复杂性。如果你不得不使用一些来自strlib.h和string.h的函数，就必须将两个库接口同时包含在程序中。更重要的是，你不得不同时理解这两个接口和它们所提供的函数。定义了strlib.h，你就可以在准备使用较低层的接口之前不需要学习低层接口。 511

```
/*
 * File: invert.c
 * --------------
 * This file implements a function InvertName(result, name)
 * that takes a name in standard order (first middle last) and
 * returns a new string in inverted order (last, first middle),
 * which makes it easier to alphabetize the names. The test
 * program reads in names and prints out their inverted form,
 * stopping when a blank line is entered.
 */

#include <stdio.h>
#include <string.h>
#include "genlib.h"
#include "simpio.h"

/*
 * Constant
 * --------
 * MaxName -- Maximum number of characters in a name
 */

#define MaxName 40

/* Private function prototypes */

static void InvertName(char result[], char name[]);

/* Main program */

main()
{
    char *standardName;
    char invertedName[MaxName+1];

    printf("This program converts a name in standard order\n");
    printf("into inverted order with the last name first.\n");
    printf("Indicate the end of input with a blank line.\n");
    while (TRUE) {
        printf("Name: ");
        standardName = GetLine();
        if (strlen(standardName) == 0) break;
        InvertName(invertedName, standardName);
        printf("%s\n", invertedName);
    }
}
```

图14-4 invert.c

```
int StringCompare(string s1, string s2)
{
    return (strcmp(s1, s2));
}
```

14.3.2 strlib分配函数的实现

strlib.h接口里大多数函数的主要特征是它们动态地为新创建的字符串分配内存。为了集中这一阶段的操作，strlib.c程序定义了一个专用函数CreateString(len)，用于给长度为len的字符串分配内存，而且还考虑到在字符串最后为'\0'预留一个额外的字节。和大多数软件包一样，CreateString的实现可以写为：

```
static string CreateString(int len)
{
    return ((string) GetBlock(len + 1));
}
```

一旦有了这个函数，你可以用它来写strlib中所有需要分配内存的函数。例如，考虑如下CharToString的实现，它为一个只含一个字符的字符串分配内存：

```
string CharToString(char ch)
{
    string result;

    result = CreateString(1);
    result[0] = ch;
    result[1] = '\0';
    return (result);
}
```

函数动态地分配空间来存放只有一个字符的字符串，然后通过赋入特定的字符和终止空字符来初始化该内存。

512 ~ 514

让我们看一个更实质性的函数，如Concat。一种方法是显式地编写所有关于复制的代码。该实现先为连接后的字符串分配空间，再依次从每个字符串复制字符，如下所示：

```
string Concat(string s1, string s2)
{
    string s;
    int len1, len2, i;

    len1 = strlen(s1);
    len2 = strlen(s2);
    s = CreateString(len1 + len2);
    for (i = 0; i < len1; i++) s[i] = s1[i];
    for (i = 0; i < len2; i++) s[i + len1] = s2[i];
    s[len1 + len2] = '\0';
    return (s);
}
```

尽管这个实现能工作，但它并没有利用较低层库中的函数。通常，ANSI字符串库中函数的设计应使得它们尽可能利用所运行的计算机的硬件特征而有效地工作。如果在你的实现中使用了这些函数，那么产生的代码常常会比采用显式字符串操作的程序更有效。所以，将strcpy作为实现的一部分使用是很有意义的，程序如下：

```
string Concat(string s1, string s2)
{
```

复习题

1. 本章要求你从三个不同的角度考虑字符串。这三个角度是什么？

2. 根据基本表示方法，怎样确定一个字符串的末尾？

3. 若将字符串看成字符数组的话，你会使用什么习语来处理每一个字符？如果将字符串看成指向字符的指针的话，习语又会发生什么改变？

4. 判断题：将参数声明为字符数组和将其声明为指针是等效的。

5. 判断题：将局部变量声明为字符数组和将其声明为指针是等效的。

6. C语言对数组变量的使用有何限制？

7. 为什么将指向局部数组的指针作为函数返回值是错误的？

8. string.h和strlib.h接口所表示的抽象模型有哪些主要区别？

9. 如果调用strcpy(s1, s2)，哪个参数是源字符串，哪个参数是目标字符串？

10. 作为客户，在调用strcpy之前应该先做什么？

11. *缓冲区溢出*是什么意思？

12. 限制函数strncpy使用的两个因素是什么？

13. 使用ANSI字符串库时，怎样达到使用StringEqual函数的效果？

14. strlib.h中的哪两个函数是作为转换函数实现的？将这两个函数包含在接口中有什么重要意义？

15. 为什么在实现Concat时，使用函数strcpy比用for循环语句更好？

程序设计练习

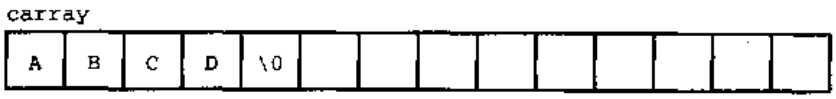
1. 字符串库中最简单的函数是strlen，它通过寻找标志字符串结束的空字符来计算字符串的长度。编写一个strlen的实现，它以一个字符数组为参数。 517

2. 重写练习1的实现，使strlen的实现以一个指向字符的指针作为参数，而且只采用指针操作来查看字符串中的每一个字符。

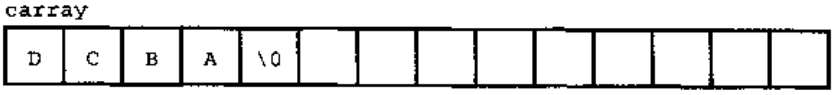
3. 写一个strcmp函数的实现，使其不调用其他函数，直接使用参数的基本表示。作为附加练习，以两种方式实现该函数，一种方式是只使用数组操作，另一种方式是只使用指针操作。

4. 第9章中的9.4.7节定义了函数Acronym(s)，该函数通过组合字符串s中每个单词的首字母来返回一个新的词，假定字符串s中包含的单词都用一个空格分隔。函数Acronym的设计是符合strlib.h模型的，因为它分配并返回了一个新的字符串。为了满足string.h接口的规则，这样的函数必须通过将数据写入调用函数提供的字符串缓冲区来传递结果。设计并实现一个采用此策略来计算词首字母的函数。你的函数不应包含strlib.h或者使用它的函数。

5. 写一个函数ReverseSting(carray)，使参数中的字符倒序排列。例如，如果carray原来包含的数据为：



调用ReverseString(carray)后，应该将字符倒序排列，使数组内容变为：



```
The word now looks like this: ----
You have 4 guesses left.
Your guess: U↵
That guess is correct.
The word now looks like this: -U--
You have 4 guesses left.
Your guess: S↵
There are no S's in the word.
The word now looks like this: -U--
You have 3 guesses left.
Your guess: N↵
That guess is correct.
The word now looks like this: -UN-
You have 3 guesses left.
Your guess: K↵
That guess is correct.
The word now looks like this: -UNK
You have 3 guesses left.
Your guess: H↵
There are no H's in the word.
The word now looks like this: -UNK
You have 2 guesses left.
Your guess: B↵
There are no B's in the word.
The word now looks like this: -UNK
You have only one guess left.
Your guess: J↵
That guess is correct.
You guessed the word: JUNK
You win.
```

图 14-5 (续)

520
~
521

```
What's in a name?
That which we call a rose
By any other name would smell as sweet.
```

文件可被视作一种二维结构，即由单个字符组成的行序列。例如，假设在你的语录集合中加入了一个新文件witches.txt，它包含出自《麦克白》的以下两行语句： 524

```
Double, double toil and trouble:
Fire burn and cauldron bubble.
```

大家很容易就认为witches.txt是由两行组成的。但从内部来说，文本文件是由一维的字符序列表示的，除了可见的打印字符外，文件还包含从第2章就开始使用的行终止符'\n'。所以，更准确地来说，文件witches.txt是以如下形式出现的：

```
Double, double toil and trouble:\nFire burn and cauldron bubble.\n
```

文本文件和字符串在很多方面有相似之处。文本文件和字符串一样，是由有序的字符集合组成的。而且，它们都有确定的结束点。在字符串中，数据结束是用空字符表示的；文本文件的结尾用一种特殊的文件结束标志，说明该标志之后再没有字符出现了。例如，witches.txt中bubble一词后的字符'\n'表示文件结束。当从某一文件中读取字符时，必须检测这个文件结束标志，从而来确定何时没有字符可读。

另一方面，字符串和文件有一些重要的区别。其中，最重要的区别就是数据能否被永久保存。字符串仅在程序运行的一段时间内临时存储在计算机内存中，而文件能被长期保存在存储设备中，直到被明确地删除或覆盖为止。使用字符串和文件中的数据的方式也是不同的。字符串是一个字符数组。指明适当的下标后，就可以以任何次序选择字符。在程序中，文件通常是顺序读写的。当程序读取一个已有文件时，会从头到尾读取字符。当程序创建新文件时，会从第一个字符开始，按照字符顺序逐个写入字符。

15.2 C语言中文件的使用

在ANSI C中，基本的文件操作是标准I/O库接口stdio.h的一部分。从第2章开始，已使用了标准I/O库，但相对而言很少有机会研究包含于其中的各种功能。标准I/O库的真正强大之处在于它允许你以可移植的方式指定文件操作，这种方式是方便、有效的。 525

在C程序中使用文件，需要完成以下工作：

1) 声明一个FILE*类型的变量。

2) 通过调用fopen函数将此变量和某实际文件相联系。这一操作称为**打开**（open）文件。打开一个文件要求指定文件名，并且指明该文件是用于输入还是输出。

3) 调用stdio.h中适当的函数完成必要的I/O操作。对输入文件来说，这些函数从文件中将数据读取至程序中；对输出文件来说，函数将程序中的数据转移到文件中去。

4) 通过调用fclose函数表明文件操作结束，这一操作称为**关闭**（close）文件，它断开了FILE*变量与实际文件间的联系。

接下来的四节将依次讨论以上的每个步骤。

```
while (TRUE) {
    printf("Input file name: ");
    filename = GetLine();
    infile = fopen(filename, "r");
    if (infile != NULL) break;
    printf("Can't open the file %s. Try again.\n", filename);
}
```

15.2.3 执行I/O操作

一旦打开一个文件，下一步要做的就是读出或写入实际数据。要完成这项工作，应该根据应用选择一种策略。最简单的方法就是使用getc和putc函数，逐个字符地读写文件。但 527
很多情况下，逐行处理文件更加方便。为了达到此目的，stdio.h接口提供了fgets和fputs这两个函数，但使用simpio.h所提供的ReadLine函数会更简单，该函数能解决使用ANSI标准函数时产生的很多问题。在更高层次上，可以选择使用fscanf和fprintf读写格式化数据，这样就可以将数字数据和字符串以及其他数据类型混合起来。上面提到的函数以后在本章还会再提到。之所以这样组织讨论是为了把基本规则相同的函数放在一起考虑。

15.2.4 关闭文件

不管选择哪种I/O操作策略，都必须确保关闭所有打开的文件。关闭文件通过调用fclose函数并配合适当的文件指针来完成。所以，关闭和变量infile、outfile相关的文件可用以下代码：

```
fclose(infile);
fclose(outfile);
```

尽管退出程序时会自动关闭所有打开的文件，但养成显式地关闭文件的习惯还是很有好处的，因为

- 程序的读者很容易知道什么时候文件在使用，什么时候不再需要使用了。
- 更容易将程序组合到自己开关文件的较大程序中。

15.2.5 标准文件

标准I/O库定义了三个特殊的标识符：stdin、stdout和stderr，它们作为FILE*常量可被所有程序使用，称为**标准文件**（standard file）。常量stdin指定标准输入文件，它是用户的输入源。常量stdout指出标准输出，表示写给用户的输出数据所在的设备。常量stderr代表标准错误文件，用于报告任何用户应该看到的出错消息。通常，标准文件都指向计算机控制台。当从stdin读取数据时，输入来源于键盘；当向stdout或stderr写入数据时，输出出现在屏幕上。但有些系统为了方便起见可以改变这些联系，所以标准输入会来源于某一文件，或者标准输出被分配给一些其他的文件或设备。

即使你所使用的系统不具备改变标准文件分配的能力，标准文件的存在还是有益处的，因为这表明了任何在本章描述过的函数都可以直接用控制台操作。使用stdin、stdout和stderr这些名字，你可以直接读写字符、行或是格式数据。 528

15.3 字符I/O

文件处理的最简便的方法是逐字符地遍历该文件。stdio.c接口定义了getc(infile)函数，该函数从某一文件中读取下一个字符，并将其返回给调用函数。同时还定义了getchar()函数，该函数从标准输入文件中进行读取，因此与getc(stdin)作用相当。

```
 * Usage: CopyFile(infile, outfile);
 * ---------------------------------
 * This function copies the contents of infile to outfile. The
 * client is responsible for opening these files before calling
 * CopyFile and for closing them afterward.
 */

static void CopyFile(FILE *infile, FILE *outfile)
{
    int ch;

    while ((ch = getc(infile)) != EOF) {
        putc(ch, outfile);
    }
}

/*
 * Function: OpenUserFile
 * Usage: fileptr = OpenUserFile(prompt, mode);
 * --------------------------------------------
 * This function prompts the user for a file name using the
 * prompt string supplied by the user and then attempts to
 * open that file with the specified mode. If the file is
 * opened successfully, OpenUserFile returns the appropriate
 * file pointer. If the open operation fails, the user is
 * informed of the failure and given an opportunity to enter
 * another file name.
 */

static FILE *OpenUserFile(string prompt, string mode)
{
    string filename;
    FILE *result;

    while (TRUE) {
        printf("%s", prompt);
        filename = GetLine();
        result = fopen(filename, mode);
        if (result != NULL) break;
        printf("Can't open the file \"%s\"\n", filename);
    }
    return (result);
}
```

图 15-1 (续)

copyFile中的while循环非常常用，值得我们仔细思考。while循环的判断表达式通过嵌入式赋值语句将读入字符和检测文件结束标记的操作结合起来。当程序判断while循环条件时，首先从判断子表达式

```
ch=getc(infile)
```

开始，该表达式读入一个字符并将其赋给ch。在执行循环体之前，程序不能确保所赋的值不是EOF。赋值语句外的括号是必不可少的，否则该表达式将把character和EOF的比较结果赋值给ch，这是不正确的。这一循环的作用与如下所示的长表达式是相同的：

```
while (TRUE) {
```

此时，tempfile已经获得所需的最终内容。如果删除原有的名为witches.txt的文件，并 532
将tempfile重新命名为witches.txt，就得到了所需的唯一一个文件，它包含我们所希望得到的内容，如右图所示。在编写实现这一策略的代码时，需要使用三个定义在stdio.h头文件中的函数——tmpnam、remove和rename，这三个函数将在下文中进行描述。

witches.txt

```
DOUBLE, DOUBLE TOIL AND TROUBLE:
FIRE BURN AND CAULDRON BUBBLE.
```

虽然我们可以为临时文件取任何名字，但stdio.h接口中包括的函数tmpnam可以为临时文件自动生成文件名。文件命名的习惯因不同机器而异。调用函数tmpnam(NULL)会返回一个字符串，它的值适合作为该台机器上临时文件的文件名[⊖]。因此，可以按照如下代码创建并打开一个新的临时文件：

```
temp=tmpnam(NULL);
outfile=fopen(temp, "w");
```

删除一个文件只需调用函数remove(*name*)即可，此处的*name*是一文件名。重命名一个文件同样非常简单，可以通过调用函数rename(*old name*, *new name*)完成。和ANSI库中的很多其他函数一样，remove和rename在调用成功后返回0，调用失败后返回一个非零值。虽然有时会遇到将这些函数当作过程使用的情况，但检验返回值以确保调用成功是更安全的一种做法。

这三个函数提供了编写程序ucfile.c（如图15-2所示）所需的内容，该程序可将一个文件中的字符转换为大写字符。

```
/*
 * File: ucfile.c
 * --------------
 * This program updates the contents of a file by converting all
 * letters to upper case.
 */

#include <stdio.h>
#include <ctype.h>
#include "genlib.h"
#include "simpio.h"

/* Private function prototypes */

static void UpperCaseCopy(FILE *infile, FILE *outfile);

/* Main program */

main()
{
    string filename, temp;
    FILE *infile, *outfile;

    printf("This program converts a file to upper case.\n");
```

图15-2 ucfile.c

⊖ 当函数tmpnam以NULL作为参数调用时，返回一个指向标准I/O库实现专用的内存的指针。因此在结束第一个临时文件之前生成第二个临时文件是不安全的。如果必须这样做，可以用CopyString或strcpy将tmpnam的结果复制到另外的内存中。

们全部都“推回”到输入文件。所幸的是，能够推回一个字符就已经可以满足绝大多数情况的需要。

为了理解函数ungetc是如何使用的，假设需要编写一个程序，将程序从一个文件复制到另一个文件，在此过程中删除所有的注释语句。众所周知，在C语言中，一条注释语句以字符序列/*开始，以序列*/结束的。删除注释语句的程序必须能在检测到开始标记/*前复制字符，在检测到后逐一读字符但不进行复制，直至检测到结束标记*/为止。

这个问题的难点在于注释标记是由两个字符组成的。如果每次从文件中复制一个字符，那么当遇到“/”时应该如何处理呢？它有可能是注释语句的开始，这时就不能将其复制到输出文件中。另外，它也有可能是除号。只有读入下一个字符后我们才能进行判断，如果下一个字符是“*”，那么要将两个字符都忽略掉，并标识出已经进入注释语句。如果不是*，则需将其推回到输入流中，留待下一个循环周期中再复制它。图15-3中所示的函数CopyRemoving-Comments实现了这一策略。

533 ~ 535

```
static void CopyRemovingComments(FILE *infile, FILE *outfile)
{
    int ch, nch;
    bool commentFlag;

    commentFlag = FALSE;
    while ((ch = getc(infile)) != EOF) {
        if (commentFlag) {
            if (ch == '*') {
                nch = getc(infile);
                if (nch == '/') {
                    commentFlag = FALSE;
                } else {
                    ungetc(nch, infile);
                }
            }
        } else {
            if (ch == '/') {
                nch = getc(infile);
                if (nch == '*') {
                    commentFlag = TRUE;
                } else {
                    ungetc(nch, infile);
                }
            }
            if (!commentFlag) putc(ch, outfile);
        }
    }
}
```

图15-3 CopyRemovingComments

15.4 面向行的I/O

由于文件通常被划分成行，因此很有必要一次读入整行数据。stdio.h中执行这一操作的函数为fgets，其函数原型为：

```
string fgets(char buffer[], int bufSize, FILE *infile);
```

此函数的作用在于将下一行文件读入字符数组buffer中。通常，fgets在读入第一个换行

同的内存，因此两者的值是相同的字符串。

常见错误 如果使用同一个字符缓冲区从同一个文件中读入多行，要记住在读入下一行数据之前，需要将数据从缓冲区中复制到其他存储空间。如果不进行这样的操作，前一行的内容将被覆盖。

解决这个问题的办法之一是利用strlib库中的函数CopyString将每一个字符串写入各自的堆内存中，如下所示：

```
void ReadTwoLines(FILE *infile)
{
    string line1, line2;
    char buffer[MaxLine];

    fgets(buffer, MaxLine, infile);
    line1 = CopyString(buffer);
    fgets(buffer, MaxLine, infile);
    line2 = CopyString(buffer);
    . . . (其余程序) . . .
}
```

另外一个解决方案就是利用在simpio.h中定义的函数ReadLine，它也可以避免与fgets相关的其他问题：

- 难以确定缓冲区的大小。有些文件中的行比较长，因此很难选择一个适合所有文件的缓冲区的大小。
- 难以判断是否超出了缓冲区的界限。如果在调用函数fgets的过程中给定一个最大值，则意味着数据不会被写到所分配的缓冲区之外，但我们同样需要知道fgets是否读入了一个完整的行。使用fgets时的唯一做法是浏览缓冲区内的所有字符，检测是否包括换行符。
- 而fgets存储换行符通常也会引起麻烦。在大多数的程序中，换行符只是一行结束的标记，而并不真正是数据的一部分。使用fgets意味着还需采取其他的步骤将缓冲区中的换行符删除。

函数ReadLine与第2章中介绍的函数GetLine很相似，唯一的区别在于其输入数据来自作为参数传递的一个数据文件。与fgets相比，ReadLine具有以下优点：

- ReadLine在需要时自动分配堆内存，使得缓冲区不可能溢出。
- ReadLine删除了标记每一行结束的换行符，所返回的数据只包括这一行中的字符。
- ReadLine返回的每一个字符串都保存在各自的内存中，因此在存储一个字符串之前不必考虑是否需要进行复制。

函数ReadLine在遇到文件结束符时返回NULL。 538

15.5 格式化I/O

在标准I/O库提供的操作中，格式化I/O函数printf和scanf最能体现C语言的特征。从第2章起我们已经开始使用函数printf，但对其功能的了解还只停留在表面上。接下来几个小节将介绍关于printf/scanf函数的更多内容，以便读者能够更有效地利用它们进行编程，以及更透彻地理解使用这两个函数的其他程序。

- 其他字符，这些字符必须与输入中的下一个字符相匹配。这使得程序能够检查所需的标点符号，如两个数字之间的逗号等。

常见错误 记住，scanf在控制字符串后的每个参数都必须是一个指针。对于简单变量来说，前面的&表明取的是它的地址。但如果该参数是一个字符数组，则该变量的名称已经自动解释为指针，因此无需再使用&。

转换说明在结构上与printf相似，但可用的选项是不同的。scanf的转换说明由以下的几种选项组成，必须遵循下面的顺序：

- 由星号（*）表示的赋值屏蔽标志，该标志表明不将输入流中的值读入参数指定的地方。
- 一个可选的数字字段宽度，指出该字段可以读入的最大的字符个数。
- 大小的说明，h表示short类型的整数值，l表示long类型的整数或double类型的浮点数。
- 转换说明符，通常为表15-1中所示的某一种说明符。

scanf的三种形式都会返回成功转换后的数据，而忽略由*号屏蔽掉的数据。如果在转换之前读到文件结束标志，则scanf返回EOF。 540

表15-1 scanf的转换说明

代 码	说 明
%d	以十进制整数的形式读入下一个值，并将该整数值存入下一个指针参数指定的存储单元。变量的大小应与说明指定的大小相匹配
%f %e %g	以浮点数的形式读入下一个值，并将该浮点数存入下一个指针参数指定的存储单元。如果指针类型的目标为float，则转换说明应为%f，如果指针类型的目标为double，则转换说明应为%lf，%e和%g与%f是相同的，它们是为了与printf函数相对应而设立的
%c	读入下一个字符，并将其存入下一个参数指定的地址，该参数必须是一个字符指针。与其他转换说明不同的是，%c在转换时不会跳过空白字符
%s	读入一个字符串，存入下一个参数指定的字符数组的连续元素中。调用程序必须确保该字符数组有足够的空间存储读入的值。遇空白字符时停止读入
%[...] %[^...]	转换说明可由一对方括号[]及其括住的字符集组成。在这种情况下，读入一个字符串并将其存入下一个参数指定的地址中，直到遇到不包含在字符集中的字符时停止读入。如果字符集由^开始，则读入字符集以外的字符。例如，转换说明为%[0123456789]，则读入由数字组成的字符串；如果转换说明为%[^.!?]，则读入的字符不包括句号、惊叹号及问号
%%	不进行任何转换，表示读入一个百分号

15.5.3 用scanf读入字符串

表15-1中包含了几个用于读入字符串的转换符——%s、%[...]和%[^...]。使用这几个转换说明时，调用程序必须负责分配足够的空间来存放数据。因此，scanf中用来存放字符串的参数应该为调用程序声明的一个数组。例如，为了读入一个由空白字符界定的字符串，必须事先声明一个数组来存放字符串，声明如下： 541

```
#define MaxWord 25
char word[MaxWord];
```

一旦获得用于存放结果的空间，程序就可以通过下面的语句读入字符串了：

```
fscanf(infile, "%s", word);
```

要注意，在变量名的前面没有&字符。函数fscanf要求它的所有参数都为指向存储空间的指

```
    if (nscan != 3 || termch != '\n') Error("Bad input line");
}
```

15.5.4 格式化I/O的一个实例

为了更好地理解scanf的工作原理，最基本的方法是研究一个使用scanf的实例，然后自己写一个类似的程序。例如，需要从文件elements.dat中读入数据，该文件为每一个化学元素列出下列信息：

- *元素名称*，通常不超过15个字符。
- *化学符号*，通常不超过2个字符。
- *原子序号*，表示原子核中质子数的一个整数。
- *原子量*，一个浮点数（用来表示该元素的各同位素的平均质量，但具体值对本例没有影响）。

543

文件的前十行如下所示：

```
Hydrogen, H, 1, 1.008
Helium, He, 2, 4.003
Lithium, Li, 3, 6.939
Beryllium, Be, 4, 9.012
Boron, B, 5, 10.811
Carbon, C, 6, 12.011
Nitrogen, N, 7, 14.007
Oxygen, O, 8, 15.999
Fluorine, F, 9, 18.998
Neon, Ne, 10, 20.183
```

假设现在要从上面的文件中读取数据，并以下列表格形式输出：

```
      Element (symbol)     Atomic Weight
-   ----------------------------------
  1. Hydrogen (H)              1.008
  2. Helium (He)               4.003
  3. Lithium (Li)              6.939
  4. Beryllium (Be)            9.012
  5. Boron (B)                10.811
  6. Carbon (C)               12.011
  7. Nitrogen (N)             14.007
  8. Oxygen (O)               15.999
  9. Fluorine (F)             18.998
 10. Neon (Ne)                20.183
```

在输出中，首先出现原子序号，然后是元素名称、化学符号，这两个内容显示在一个固定宽度的字段中，最后是原子量。

第一步是为fscanf设计一个控制字符串，以便正确地从文件中读入一行。输入行由元素名称开始，元素名称是用逗号标志结束的字符串。最简单的方法是利用转换符%15[^,]来读入第一个逗号前出现的字符。字段宽度为15保证了elementName的缓冲区不溢出。控制字符串中在此转换符后应该跟一个逗号，与输入流中的内容相对应，此后为一个空格，用来跳过逗号后面的所有空格。化学符号通过转换说明%2[^,]来读入，情况与前一个转换说明基本相同，只是字段宽度较小。最后两个字段分别为int和double型的数值。为读入这些内容，

```
    infile = fopen(ElementsFile, "r");
    if (infile == NULL) Error("Can't open %s", ElementsFile);
    printf("     Element (symbol)     Atomic Weight\n");
    printf("-----------------------------------\n");
    while (TRUE) {
        nscan = fscanf(infile, "%15[^,], %2[^,], %d, %lf%c",
                               elementName, elementSymbol,
                               &atomicNumber, &atomicWeight,
                               &termch);
        if (nscan == EOF) break;
        if (nscan != 5 || termch != '\n') {
            Error("Improper file format");
        }
        sprintf(namebuf, "%s (%s)", elementName, elementSymbol);
        printf("%3d. %-20s %8.3f\n", atomicNumber, namebuf,
                                     atomicWeight);
    }
}
```

图 15-4（续）

15.5.5 使用scanf的局限

scanf的优势在于为读取固定格式的输入提供了一个简便的方法。通常只要确定一个正确的转换说明，就可通过调用一个函数来读取很多数据项。特别是需要编写测试应用的代码并且读入一些测试数据时，scanf不失为一个很好的工具。在这种情况下，测试数据及其格式都应该是已知的。

但是，scanf函数往往对于无法控制来源的输入流束手无策。如果输入数据中可能存在错误，程序就必须测试这些输入数据，确保它具有正确的格式。但是，使用scanf时通常不可能彻底检查输入数据。在商业化的C程序中，错误检测是极其重要的一个环节，因此scanf 546
并没有得到广泛的使用。对ANSIC库进行标准化的委员会主席P.J.Plauger给出了这样的提示：

> 你会发现输入的转换说明并不像输出的转换说明那样完整。很多时候，你希望控制输入。或者你会发现不可能确定哪里出现了输入错误，并适当地恢复错误……。随时准备……在某一时刻放弃输入函数吧。多年的经验告诉我们，它们的用途实在有限[⊖]。

小结

如果想在一个程序执行结束后永久地保存信息，则必须将其存入支持永久存储的辅助存储设备（如磁盘等）上的文件中。程序设计中最常用的文件形式为文本文件，这种文件由字符数据组成。文本文件可以用来进行输入和输出，且通常按顺序进行处理。

stdio.h接口提供了多种用于输入/输出操作的函数，可根据不同需要进行选择。例如，可以逐字符地处理文件，也可以逐行处理，还可以按照一组格式化的字段进行处理。

本章的重点内容包括：

- 要在程序中使用文本文件，首先需要声明一个FILE*类型的变量来保存系统追踪对该文件进行的操作时所需的信息。

⊖ P.J.Plauger, *The Standard C Library*, Englewood Cliffs, NJ: Prentice Hall, 1992, p. 268。

程序设计练习

1. 写一个程序wc.c，要求读入一个文件，并报告该文件中有多少行、多少个字和多少个字符。在本程序中，字由连续的字符组成，以空白字符为界。例如，文件lear.txt中包含《李尔王》中的下列段落：

```
Poor naked wretches, wheresoe'er you are,
That bide the pelting of this pitiless storm,
How shall your houseless heads and unfed sides,
Your loop'd and window'd raggedness, defend you
From seasons such as these? O, I have ta'en
Too little care of this!
```

执行程序后应该得到下面的结果：

```
File:  lear.txt↵
Lines:    6
Words:   43
Chars:  254
```

2. 出版商发现有时不必考虑文字，仅仅对文字布局及版式设计进行考察是有好处的。为了达到这个效果，出版商会对部分样张进行排版，将原来的字母替换成随机选择的字母。这样得到的文本在格式和标点上与原文完全相同，只是不再表达原来的意思。在出版业中，这样的文本替换被称为greek，有可能出自于Julius Caesar中的一句谚语“It's all Greek to me”（我对此一窍不通）。 549

写一个程序greek.c，要求能够从一个输入文件中读取字符，并对字符进行适当的随机替换，最后输出至控制台。输入中的大写字符应被任何随机的大写字符替换，而小写字符则被随机的小写字符替换。对于非字母的字符不进行替换，照原样输出。例如，如果输入文件troilus.dat是从Troilus and Cressida中节选的：

```
Ay, Greek; and that shall be divulged well
In characters as red as Mars his heart
Inflamed with Venus:
```

那么程序的输出结果应为：

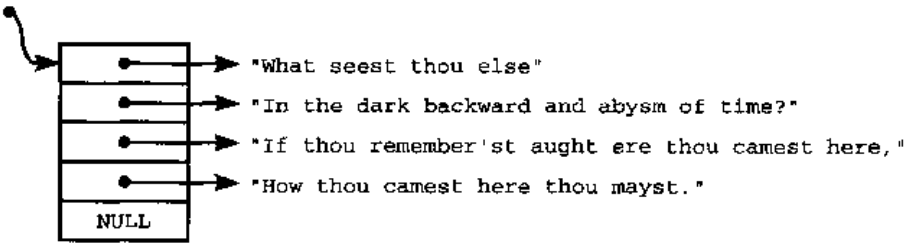
```
Hb, Jwyqt; cwq ocgs lsosn jo hdricyoc lino
Dm bzongsmdrv uj qew ya Okor umj ioyvq
Ipqpqnpj vrvy Snszv:
```

3. 一些文件运用制表符将数据按列对齐。但是对那些不支持制表符的应用来说，这种做法会导致错误。对于这些应用来讲，需要一个程序来将输入文件中的制表符转换为相应个数的空格。在程序设计中，制表符通常为8位。例如，假设输入文件中有这样一行表格：

```
abc⟶| nopqr⟶| xyz
```

```
What seest thou else
In the dark backward and abysm of time?
If thou remember'st aught ere thou camest here,
How thou camest here thou mayst.
```

如果将prospero.txt作为infile的输入打开，并调用函数ReadFile(infile)返回一个指向动态字符串数组的指针，如下面的图所示：



在设计ReadFile的实现时，最重要的一个问题是应该为动态数组分配多少空间。因为在读一个文件之前，我们无法知道其中包含有多少行的内容。解决这个问题最常用的方法是为数组任意指定一个初始大小，之后如有必要再随时增加新的空间。如果需要分配更多空间，需要按照以下几步进行：

a. 分配一个新的动态数组，该数组要比原来的数组大。

b. 将旧数组的内容复制到新的数组中。

c. 释放旧的数组。

d. 将新数组的指针赋给指向数组的变量。

将ReadFile的实现写入一个单独的readfile.c文件中，并设计一个readfile.h接口，使客户可以调用这个接口。用readfile.h接口写一个主程序将文件中的行按倒序输出。例如，以prospero.txt中的内容作为输入，则程序的输出应为：

```
How thou camest here thou mayst.
If thou remember'st aught ere thou camest here,
In the dark backward and abysm of time?
What seest thou else
```

552

6. 在20世纪60年代，演员Steve Allen经常在他的喜剧表演中玩一个名为Madlibs的游戏。Allen让观众提供某一类型的词，如动词、形容词、或复数名词等，并将这些词填入事先准备好的文本的空格中，再将它们念给观众听。

为了说明这一过程，假设事先准备好的文章是从《哈姆雷特》中节选的，并已将几个关键字删去了，如下所示：

To (动词) or not to (也是动词): that is the question.

Whether 'tis nobler in the mind to suffer

The slings and arrows of (形容词) fortune,

Or to take arms against a sea of (名词复数),

And by opposing end them?

观众选择适当类型的词将空格填满，但事先并不知道原文是什么。按照所选的词，得到结果可能是相当可笑的，尽管形容词silly可能用的很多。

注意，程序将单词中的所有字符都转换为小写，因此Some和some不会被作为不同的词处理。

在设计解决方案时，要特别考虑模块的分解。什么样的模块有助于解决这个问题？特别是有没有现有的模块可以应用？

8. 写一个程序，将包含你所要购置计算机设备的清单的文件转换成更优美的格式。输入文件由若干行信息组成，每一行有以下几个字段：

a. *零件编号* 最多由六个字符组成，从每一行的开始至第一个空格处为止。 554

b. *产品名称* 最多由20个字符组成。这一字段从零件编号后开始，一直到第一个“/”字符为止，该字符必须出现在每行中。

c. *产品数量* 为一整型值。

d. *单位价格* 一个浮点型数值，通过@符号与产品数量进行区分。

在字段与字段之间可出现空白字符。但每一行中都要包含这四个字段的数据以及所需的区分标志。输入文件的格式如下面的文件order.dat所示：

```
NC271X Notebook computer / 1 @ 1729.00
LP552 Laser printer / 1 @ 1499.00
TC552V Toner cartridge / 2 @ 99.00
DS420 3.5" diskettes / 10 @ 2.99
CC542 Carrying case / 1 @ 27.99
```

程序应该读入该输入文件，并以固定宽度显示每个字段。另外，程序需计算出每一行货品的总价（产品数量×单位价格），以及整个订单的总额。按照上面给出的数据，程序的结果应如下所示：

```
Order file: order.dat↵
NC271X    Notebook computer      1 @ 1729.00 = 1729.00
LP552     Laser printer          1 @ 1499.00 = 1499.00
TC552V    Toner cartridge        2 @   99.00 =  198.00
DS420     3.5" diskettes        10 @    2.99 =   29.90
CC542     Carrying case          1 @   27.99 =   27.99
-------------------------------------------------------
TOTAL                                          3483.89
```

9. 利用第7章的图形库，写一个程序在屏幕上显示多边形。多边形顶点的坐标保存在一个数据文件中，每一行为一对坐标。每一个坐标由两个浮点型数值组成，由圆括号括起，中间以逗号分隔。输入文件中可能包含多个多边形的顶点坐标，每一个多边形靠一个空白行与前面一个多边形区分开来。例如，文件polygon.dat包含了一个正方形坐标和一个三角形的顶点坐标，如下所示： 555

```
(0.5, 0.5)
(1.0, 0.5)
(1.0, 1.0)
(0.5, 1.0)

(2.0, 0.5)
(3.0, 0.5)
(2.0, 1.0)
```

第16章 记　录

Let's look at the record.

— Al Smith, presidential campaign slogan, 1928

学习目标

- 理解记录的概念以及在编程中的重要作用。
- 理解定义一个记录类型和声明一个记录变量的差别。
- 能够通过“.”和“→”运算符来操控记录以及记录指针。
- 能够为新记录动态地分配存储区。
- 理解数据库的设计过程。

557

在第11章学习数组的时候，我们已经初步理解了计算机程序设计的一个重要理念：利用复合的数据结构表示复杂的信息集合。在程序中声明一个数组，即可将任意多的数据聚集起来构成一个概念上的整体。需要时可以从数组中选出元素，并单独进行操作；也可以将它们作为一个整体，同时进行操作。

现代程序设计语言的一大特点是能够将各自独立的数据组织成一个彼此相关的整体。过程和函数可以将不同的操作统一在一个名称下。在数据处理方面，复合的数据结构，例如数组，可以实现这种统一。在很多情况下，将小的程序段整合成为单一的、更高级的结构，有助于简化程序并提高程序的可读性。这种统一不是最近才出现的，这一理念已经对国家和社会行为产生了影响，如劳工赞歌中的“团结让我们更有力”，以及美国国玺上的名言“一从众中来”等。

当为现实世界中的一组数据建模时，数组是非常有用的一个工具。而这样的数据往往具备两条基本性质：首先，数据必须是有序的，因而可以通过索引号码来引用某个元素。其次，数据必须为同质的，即所有的元素的基本类型必须相同。当试图对现实世界中的一组或一系列相似的事物进行建模的时候，数组往往是最理想的工具。另一方面，将一组无序的、异质的数据看作一个整体也很重要。在C语言中，这样的一组数据被称为**结构**（structure），而在计算机科学这一领域内则被称为**记录**（record）。本书在大多数情况下使用记录一词，但需要突出C语言中对于记录的特殊实现时会改用结构一词。

16.1 数据记录的概念

为了更好地理解记录的概念，假设你负责某家小公司的薪金支付系统。你必须了解每一个雇员的信息。例如，为打印工资单，你必须知道雇员的姓名、职位、社会保障号、薪水、扣缴税款的情况，以及其他相关信息。这些信息组成了一个雇员的记录。

雇员记录是什么样子的呢？把记录想象成表格中的行是最容易理解的。例如，想象一下查尔斯·狄更斯的小说《圣诞颂歌》中斯克鲁齐（Scrooge）和马列（Marley）的小公司，并将今天的社会保障号码及扣缴税款补贴加进去，则雇员花名册包含两个记录，相关的数据如下所示：

558

声明和函数中的变量声明类似。例如，下列代码定义了名为employeeT的结构类型，用以表示雇员记录：

```
typedef struct {
    string name;
    string title;
    string ssnum;
    double salary;
    int withholding;
} employeeT;
```

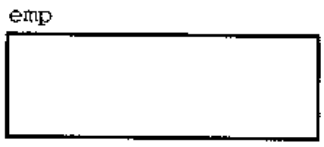
这一定义为具有新类型employeeT的所有对象提供了一个模板。每一个对象都具有五个字段，由name字段开始（该字段为字符串），直至withholding字段结束（该字段为整型数）。

16.2.2 声明结构变量

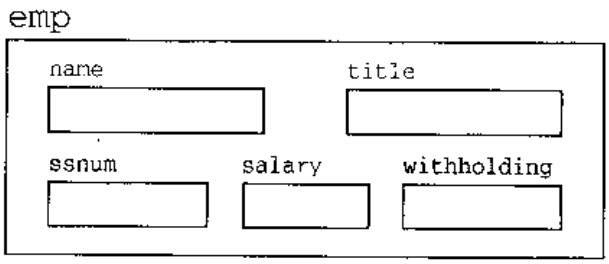
定义了一个新类型后，下面的一步就是声明该类型的变量。例如，有了employeeT这个类型，就可以通过下列代码定义该类型的一个变量emp：

```
employeeT emp;
```

如果需要用方框图表示这个变量，可以采用两种方式。如果从整体的角度来看待，即在概念上从一个较远的距离来观察，则得到右图所示的一个名为emp的方框。但如果从近处看细节，会发现标签为emp的方框内部还存在着五个单独的方框（见下图）：



560



16.2.3 记录选择

当通过如下语句声明了变量emp后，

```
employeeT emp;
```

就可以用这个名称指代该记录，但有时需要打开该记录结构并对单个字段进行操作。如果需要引用记录中的某一个字段，需要写下整个记录的名称，后面跟符号“.”以及该字段的名称。因此，为引用存储在emp中雇员的职位，需要写以下代码：

```
emp.title
```

在此，符号“.”称为“点”，因此可以将这个表达式读作“emp点title”。通过点运算符选择字段被称为**记录选择**（record selection）。

16.2.4 记录初始化

和对待其他类型的变量一样，我们可以通过赋值的方式对记录变量进行初始化。点运算符返回一个左值，表示可以对一个记录选择表达式进行赋值。例如，如果执行下列语句：

```
emp.name = "Ebenezer Scrooge";
emp.title = "Partner";
```

```
pointT AddPoint(pointT p1, pointT p2)
{
    pointT p;

    p.x = p1.x + p2.x;
    p.y = p1.y + p2.y;
    return (p);
}
```

当调用函数AddPoint(p1, p2)时，(其中p1，p2都为pointT类型的值)，函数返回的点的坐标是p1和p2相应坐标的和。

当调用参数为记录的函数时，记录参数的值就复制到相应的实参。当函数改变实参的值，或改变某个字段内容时，形参还保留着原来的值。这种处理方式和C语言处理所有参数（除数组以外）一样。

16.3 数组与记录的结合

现代程序设计语言的一大特点是，当你拥有一个类型时，就可以利用它来创建更复杂的新类型。为了达到这一目的，通常可以定义记录数组，或者包含数组的记录。同理，这样的新类型同样可以用来创建更加复杂的类型。

例如，在前面的16.2.1节中定义了一个名为employeeT的类型。一旦定义了该类型，就可以用它来声明数组，而数组的元素的类型就是employeeT。下面的代码

```
int scores[10];
```

声明scores是具有10个整型值的数组，而

```
employeeT staff[10];
```

563

就声明staff是具有10个雇员的数组。雇员Ebenezer Scrooge和Bob Cratchit的记录刚好适用于这个数组，成为staff数组的前两个元素，如下所示：

staff

0	Ebenezer Scrooge	Partner	271-82-8183	250.00	1
1	Bob Cratchit	Clerk	314-15-9265	15.00	7

可以选取staff的任意元素，就如从其他数组中选取元素一样。例如，可以用staff[1]来选取Bob Cratchit的整个记录：

Bob Cratchit	Clerk	314-15-9265	15.00	7

对于这条记录，可以进一步选取其中的字段，例如：

```
staff[1].name
```

就选取了Bob Cratchit的名字：

Bob Cratchit

选取名字字段可能已经满足了应用的需要。但也可以进一步进行选取。Name字段的值是一个字符串，因此可以继续对其中的元素进行选取。如果想知道staff数组中第一个雇员的名字的首字母，就可以用下面的语句：

```
staff[1].name[0]
```

你还可以定义适合在雇员数据库系统中使用的其他函数。

16.4 记录的指针

虽然在C语言中会直接使用一些简单的记录，但大多数情况下，存储结构化数据的变量通常被定义为记录的指针，而不是记录本身。指向记录的指针通常比记录小且较容易操作。更重要的是，当把记录的指针传递给某个过程时，过程可以改变记录的内容。如果直接传递记录，该记录将被复制（如同复制一个整型值一样），这样就不可能对其内容作出永久性的修改。在第11章中我们已经认识到，在C语言中，利用指针传递数组变量可在函数中对数组元素进行修改，同样的，传递记录的指针可以对记录中的字段进行操作。

为了练习使用记录指针，可从声明指向employeeT记录的指针入手，employeeT的定义如下：

```
typedef struct {
    string name;
    string title;
    string ssnum;
    double salary;
    int withholding;
} employeeT;
```

有了这个定义，就可以声明一个employeeT类型的指针变量，代码如下：

```
employeeT *empptr;
```

该语句说明变量empptr是一个指向employeeT类型对象的指针。进行声明后，仅仅为指针提供了空间。在使用指针empptr前，仍然需要为记录中的字段提供完整的空间。

在介绍如何为记录创建空间之前，有一个问题值得我们考虑。在前面提到的例子中，我们定义了一个代表员工的记录类型，并声明了指向该类型的指针。这样的操作让我们必须同时考虑两个不同的类型：employeeT类型，以及employeeT类型的指针。如果能够单独采用一个具有指针优点的简单类型，那么我们的操作会变得更容易。

16.4.1 定义一个指向记录类型的指针

在前面的一节中提到，employeeT类型被定义为一个记录，因此在声明该类型的指针时前面必须加一个*。在大多数情况下，可以采用另外一种声明策略。如果你希望通过地址访问记录类型的值，可以将employeeT定义为一个指针而不是一个记录。我们只需对前面employeeT的定义做出一个微小的改动，即可达到这一目的。如下所示： 566

```
typedef struct {
    string name;
    string title;
    string ssnum;
    double salary;
    int withholding;
} *employeeT;
```

此类型定义与前面的定义的唯一差别就在于employeeT前面的星号。这个定义表明employeeT类型为指向所说明记录的指针，虽然在定义中说明的记录没有一个明确的名称。

在建立了这个新类型定义之后，就可以用与前面类似的方法来声明employeeT的对象了，声明如下：

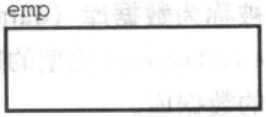
```
employeeT emp;
```

虽然使用GetBlock和sizeof的习语简单明了，但在基于记录的程序设计中为新记录分配空间的操作是十分基本的操作，因此需要更简单的方式。基于这种考虑，genlib.h接口定义了函数New，该函数以指针类型名作为参数，并返回一个指针指向足够大的空间来存储可能的数值。因此，如果有下列语句：

```
emp = New (employeeT);
```

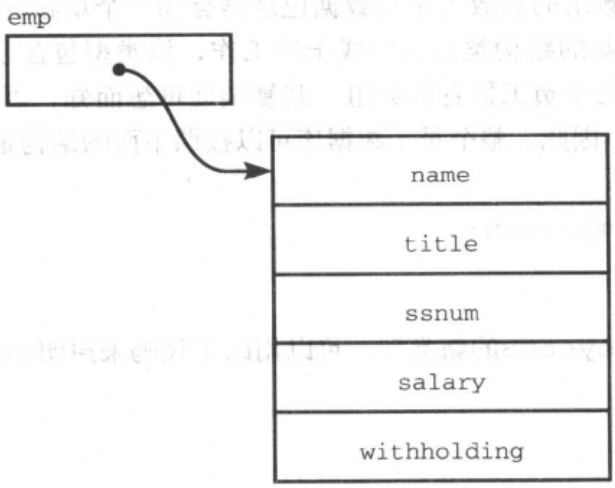
则系统会从堆中为一个员工记录分配空间，并返回一个指向该记录的指针。

这一操作的结果可以通过下面的描述来说明。在执行对New的调用前，emp中存储了一个指针，但却并未进行初始化。因而变量emp与任何记录的存储都没有关联，可以用一个空的方框表示（如右图所示）。赋值语句




```
emp = New (employeeT);
```

为记录分配了空间，并将指针的地址赋给emp，这一过程可以用下面的图表示：



16.4.3 对记录指针进行操作

虽然大多数情况下会把emp这样的变量作为一个完整的实体考虑，但有些时候我们也需要访问记录中的某个字段。在没有使用指针的记录中，我们使用点运算符来选取某个字段。当使用记录指针时，应该怎样访问某个字段呢？

在寻求答案的过程中，我们很容易被直觉误导。对于下面的声明

```
employeeT emp;
```

下面的语句是不正确的：



```
*emp.salary
```

（这是不正确的）

569

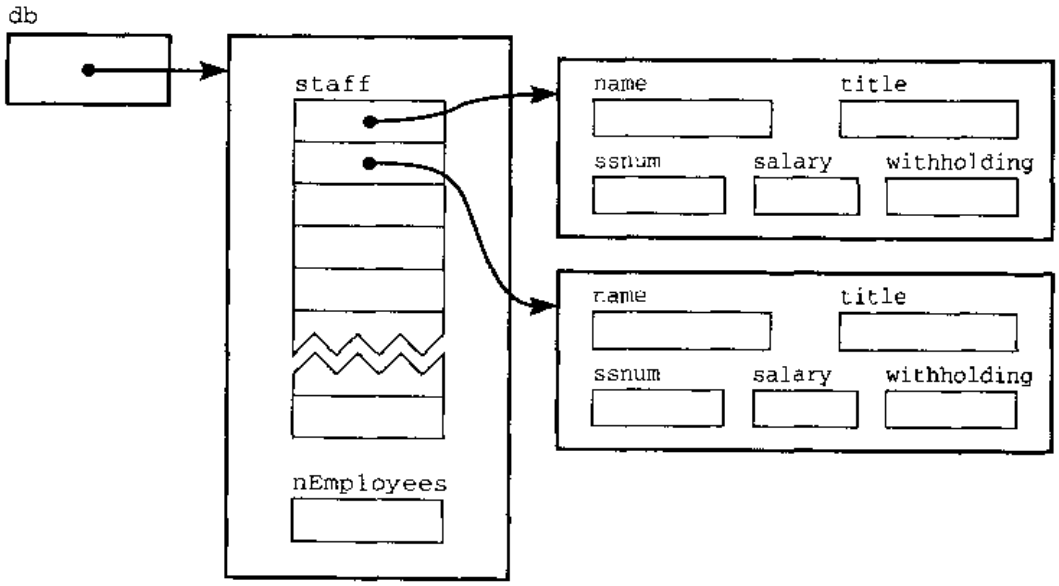
与我们的设想相反，这条语句不能选取emp所指对象的salary字段，因为这条语句中的运算符优先级不支持这样的数据解释。选择运算符的优先级高于间接引用*，因此该语句被解释为如下所示的无意义的表达式：

```
*(emp.salary)
```

而不是原本打算的：

```
(*emp).salary
```

最后，staff数组中的每一个元素（根据最近的employeeT类型定义）都是一个指向记录的指针，因此完整的图应该包含指向单独employeeT结构的指针，如下图所示： 571



为了创建这个结构，应该从部分着手。对于结构中的每一个由指针指向的方框，都需要调用函数New来为该记录分配存储空间。

为使这个例子更加具体，假设我们需要编写程序从用户处读入一个员工信息数据库。对于每一个记录，程序需要得到每个雇员五方面的信息。需要用一个空白的name字段作为数据库结束的标记。每个记录都应该存入数据库中，且要对记录进行计数。

读入数据库的操作很复杂，应该定义为一个函数。由于数据库为一个指针，ReadEmployeeDatabase只能分配指针需要的存储空间，并返回一个指针。因此，主程序声明一个数据库：

```
employeeDB db;
```

并通过调用以下函数读入值：

```
db = ReadEmployeeDatabase( );
```

ReadEmployeeDatabase的实现很简单，如果用逐步精化的方法就更简单。只需将其分解为两个函数：函数ReadEmployeeDatabase创建资料库结构；函数ReadOneEmployee读入每个员工的信息。由于函数ReadOneEmployee负责从文件中读入数据，因此该函数必须能够判断文件是否结束，并将信息通过返回一个特殊的标志传递给ReadEmployeeDatabase。由于函数ReadOneEmployee返回的值为employeeT类型的指针，因此最好用NULL作为文件结束标志。 572

函数ReadEmployeeDatabase的代码为：

```
static employeeDB ReadEmployeeDatabase(void)
{
    employeeDB db;
    employeeT emp;
    int nEmployees;

    db = New(employeeDB);
    nEmployees = 0;
```

```
            db->staff[i]->salary *= 2;
        }
    }
}
```

16.6 基于记录的应用程序设计

二十年前，当计算机刚刚作为通用工具进行普及时，在教育界掀起了一股热潮，即让计算机参与到教学的过程中。有人提出了一种技术，称为**程序化教学**（programmed instruction），即由计算机辅助的教学工具提出一系列的问题，并由前面问题的回答决定后面的问题。如果学生的回答全部正确，则计算机会跳过大部分简单的问题，直接提出难度较大的问题。而对在回答中遇到困难的学生，整个过程会进行的较慢，留出时间让学生复习总结。

设想一下，如果要求你编写一个应用程序实现程序化教学。简单来说，你的程序必须能够做到下面几点：

- 向学生提出问题。
- 从学生处获得答案。
- 继续下一个问题，根据学生的答案选择下一个问题。

这样的程序比商业化的应用程序简单很多，但从中可以体验程序设计所涉及到的基本原理。 574

16.6.1 使用数据库的重要性

将程序化教学的应用程序设计为一个过程的集合是可行的。每个过程提出一个问题，读取学生的答案，并根据学生的答案调用另外一个适当的过程。但是这样的程序很难进行改变，如果想加入新的问题或为另外一门课程设计程序，则需要重新编写过程。对于懂得编程的人来讲，写程序并不是很困难的事，但并不是所有人都懂得编程。大多数程序化教学的课程都是由某一学科的教师进行设计，而这些教师通常不是程序员，让他们来编写程序会限制应用程序的功能。

作为该项目的程序员，你的目标是开发一个应用程序，提供程序化教学，使不具备编程技能的老师可以提出问题、接受答案并根据前后对照的信息使程序按照适当的顺序进行。为了达到这个目标，可以将程序设计为一个通用的工具，该工具可以从文件中读入与程序化教学相关的信息和数据。如果你采用这种方法，只要提供不同的数据文件，同一个程序就可以支持不同的课程了。

在该应用程序的设计中，可以从以下几个具有普遍性的问题入手进行考虑，例如：

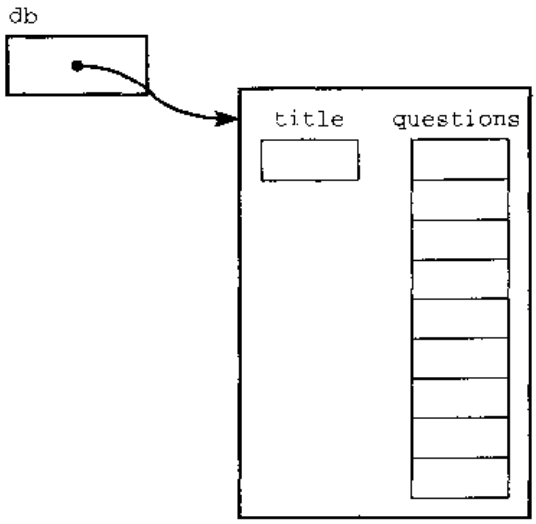
- 一般问题的要求是什么？要特别理解程序所必须支持的所有操作，而不仅仅是某一学科的要求。
- 如何在你的程序中表示程序化教学课程所需的信息？在设计阶段，你需要开发一个合适数据结构，它可以是记录和数组的结合。
- 课程的数据文件是怎样的？当你做出决定时，你必须意识到数据文件是由非程序员编辑操作的，他们的专长在于所教授的课程，而不是编程。
- 如何将数据文件使用的外部表示转换为内部表示？
- 如何编写程序对数据库进行操作？

本章的剩余部分将依次探讨这几个问题。

16.6.2 问题的框架

在某一层次上，写出程序的提纲并不困难。当程序运行时，基本的操作是重复执行以下

解应该如右图所示。



虽然现在仍然不知道用什么类型来表示问题，但仍然可以将这个图转换为一个结构定义。我们已经知道了其他字段的类型，因此可以写出一个恰当的记录定义，如：

```
typedef struct {
    string title;
    questionT questions[MaxQuestions+1];
} *courseDB;
```

常量MaxQuestions为程序所允许的问题数目的最大值。由于数组下标总是从0开始，而课程总是从问题1开始，因此需要为数组增加一个元素，使得第MaxQuestions个问题可以存在。

courseDB的定义使我们可以声明变量db来存放整个数据库，定义如下：

```
courseDB db;
```

但是这个声明仅为指针保留存储空间，因此是与前面所示的虚线框部分相应的。当在程序中创建结构时，需要为该结构分配空间，代码如下：

```
db = New(courseDB);
```

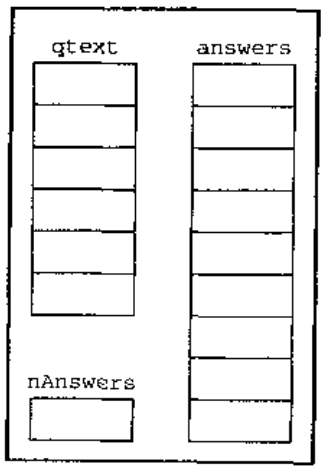
此时，需要增补一些丢失的细节。例如，我们仍然没有为courseDB定义中的类型questionT作出定义。为使数据结构完整，还需补充这方面的设计。问题由文本组成，其中包括多行文件以及一系列可能的答案。这两个结构都可由数组表示。问题的文本是字符串的数组，每一个字符串包含了一行的内容。而答案存储在一个结构较为复杂的数组中。目前，我们可以将答案声明为一个类型answerT的数组，其细节将在下一步完善。

通常情况下，我们需要提供一个机制来追踪数组的有效大小，因为它通常比所分配的空间小一些。记录有效大小可以利用下面两种方式：

- 可以在数组的最后一个值后面增加一个标记值。
- 可以将元素的个数存入一个整型变量，并将其作为记录的一部分。

究竟采用哪种方式取决于数组的使用方式，以及是否可以选择一个合适的标记值。

为了介绍这两种模式的操作，示例程序采用了两种定义questionT的方式。组成问题文本的行存入了一个由字符串组成的数组。对于指针数据来讲，常量NULL是一个最佳的标记值。而由于答案数组的元素都是answerT类型的，这种类型仍未进行定义，因此很难知道选择什么样的标记值。因此，在此例中更适合用一个变量存储答案的数量。



一个问题的结构可用右图表示。因为这个结构比较复杂，而且需要将其考虑为一个整体，因此可以将此结构定义为一个指针类型，如下所示：

```
typedef struct {
    string qtext[MaxLinesPerQuestion+1];
    answerT answers[MaxAnswersPerQuestion];
    int nAnswers;
} *questionT;
```

将此结构声明为一个指针意味着必须调用New来为每个问题分配

问题的文本由单独的一行文件构成，接下来是两个可能的选择。如果用户输入false，则程序继续提出问题2；而如果用户输入true，则程序跳至问题3。

这个例子提出了一个有趣的问题，即如何将问题序号与每一个问题联系起来？一种方法是将文件中的问题排列并依次编上序号。这种做法给程序员带来便利，却给课程设计者造成一定的困难。

如何理解这一点呢？假设课程设计者想在现有课程的开始处加入一个问题。那么，所有后续的问题都要向后移一位，所有的问题序号也要随之改变，因此课程设计者需要花很多的时间完成后续的处理工作。更好的办法是由程序编写者给每个问题编一个序号。例如，如果示例中关于地球和太阳的问题是1号的话，它的文件可以包含该问题的编号，如下图所示：

```
1
True or false: The earth revolves around the sun.
-----
true: 2
false: 3
```

由课程设计者提供问题序号的优点在于，这样会使课程设计的编辑变得相对简单。如果想要增加一个问题，只需给该问题一个从未使用过的编号就可以了。其他问题的编号无需改动。课程设计者可以在文件中的任何位置加入新的问题，因为已经无需确保问题序号的连续性了。

16.6.5 程序代码

决定了内部数据结构及外部文件格式后，编写程序的过程就变得很简单了，只需将整个任务分解成单独的函数，并对它们进行逐步精化就可以了。完整的程序在文件teach.c中，如图16-1所示。 581

```
/*
 * File: teach.c
 * -------------
 * This program executes a simple programmed instruction course.
 * The course is specified by a data file containing all the
 * course information.  The data structures and the format of
 * the data file are described in Chapter 16.
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/*
 * Constants
 * ---------
 * MaxQuestions          -- Maximum question number
 * MaxLinesPerQuestion   -- Maximum number of lines per question
```

图16-1 teach.c

```
static void ReadQuestionText(FILE *infile, questionT q);
static void ReadAnswers(FILE *infile, questionT q);
static FILE *OpenUserFile(string prompt, string mode);
static void ProcessCourse(courseDB course);
static void AskQuestion(questionT q);
static int FindAnswer(string ans, questionT q);

/* Main program */

main()
{
    courseDB course;

    course = ReadDataBase();
    ProcessCourse(course);
}

/* Section 1 -- Functions to read the data file */

/*
 * Function: ReadDataBase
 * Usage: ReadDataBase();
 * ----------------------
 * This function asks the user for a file name and reads
 * in the database for the course.  The file is formatted
 * as discussed in the section "Designing the external
 * structure" in Chapter 16.
 */

static courseDB ReadDataBase(void)
{
    FILE *infile;
    courseDB course;

    infile = OpenUserFile("Enter name of course: ", "r");
    course = New(courseDB);
    course->title = ReadLine(infile);
    while (ReadOneQuestion(infile, course));
    fclose(infile);
    return (course);
}

/*
 * Function: ReadOneQuestion
 * Usage: while (ReadOneQuestion(infile, course));
 * -----------------------------------------------
 * This function reads in a single question from infile into the
 * course data structure.  As long as the complete question is
 * read successfully, this function returns TRUE.  When the end
 * of the file is encountered, the function returns FALSE.
 * Thus, the "Usage" line above reads the entire data file.
 */

static bool ReadOneQuestion(FILE *infile, courseDB course)
{
    questionT question;
```

583

图 16-1 (续)

```
static void ReadAnswers(FILE *infile, questionT q)
{
    string line, ans;
    int len, cpos, nextq, nAnswers;

    nAnswers = 0;
    while ((line = ReadLine(infile)) != NULL
           && (len = StringLength(line)) != 0) {
       cpos = FindChar(':', line, 0);
       if (cpos == -1) Error("Illegal answer format");
       ans = SubString(line, 0, cpos - 1);
       nextq = StringToInteger(SubString(line, cpos+1, len-1));
       q->answers[nAnswers].ans = ConvertToUpperCase(ans);
       q->answers[nAnswers].nextq = nextq;
       nAnswers++;
    }
    q->nAnswers = nAnswers;
}

/*
 * Function: OpenUserFile
 * Usage: fileptr = OpenUserFile(prompt, mode);
 * --------------------------------------------
 * This function prompts the user for a file name using the
 * prompt string supplied by the user and then attempts to
 * open that file with the specified mode.  If the file is
 * opened successfully, OpenUserFile returns the appropriate
 * file pointer.  If the open operation fails, the user is
 * informed of the failure and given an opportunity to enter
 * another file name.
 */

static FILE *OpenUserFile(string prompt, string mode)
{
    string filename;
    FILE *result;

    while (TRUE) {
       printf("%s", prompt);
       filename = GetLine();
       result = fopen(filename, mode);
       if (result != NULL) break;
       printf("Can't open the file \"%s\"\n", filename);
    }
    return (result);
}

/* Section 2 -- Functions to process the course */

/*
 * Function: ProcessCourse
 * Usage: ProcessCourse(course);
 * -----------------------------
 * This function processes the course supplied by the caller.
 * The basic operation consists of a loop that (a) prints out
 * the current question, (b) reads in an answer, (c) looks up
```

586

图 16-1 (续)

```
static int FindAnswer(string ans, questionT q)
{
    int i;

    for (i = 0; i < q->nAnswers; i++) {
        if (StringEqual(ans, q->answers[i].ans)) return (i);
    }
    return (-1);
}
```

图 16-1 (续)

这个程序值得我们花时间仔细阅读，并注意下面几个问题：

1) 程序是怎样利用文件中的数据创建内部数据结构的?

2) 程序如何运用这些数据结构处理每个问题?

16.6.6 数据驱动设计方法的重要性

图16-1中所示的程序teach.c从课程数据文件中接收了所需的全部数据。该应用程序所提出的问题，接受的答案，以及问题的顺序等信息全部来自文件，而不是程序本身。根据从数据库中获得信息而运行的程序称为**数据驱动**（data-driven）程序。与将所有信息包含在程序设计过程中的程序相比，数据驱动的程序通常较短、较灵活，且较容易维护。

我们只需运行程序就可以理解teach.c这样的数据驱动系统是多么的灵活。按照这个项目的最初目标，程序teach.c应该用于传统的程序化教学，如文件cs.dat最开始的几个问 588
题可参见图16-2。当程序teach.c与这个文件结合应用时，得到的结果可参见图16-2。

由于这是一个数据驱动的程序，所以同样一个程序可应用于不同的环境中。例如，图16-3中显示了数据文件advent.dat中开始的几行，这是从Willie Crowther在20世纪70年代初开发的计算机游戏改编的。当运行示例程序时，将程序teach.c与文件advent.dat结合起来时，该程序的目的与运行cs.dat时大相径庭。虽然程序teach.c没有任何改变，但程序化教学的课程已经变为了一个游戏，而唯一的差别仅在于数据文件不同。

小结

在第11章里曾经介绍过可以用数组存储同质的、有序的数据集合。本章介绍了一个同样重要的概念——记录，它可以将无序的、异质的数据存储在一个集合中。数组通常用于为真实世界中的对象集合建模，而记录则用来为由若干个可以识别的部分组成的对象建模。同时，可以将数组和记录结合起来，组成具有任意复杂度的数据层次结构。用来表示数据集合的一个完整结构称为数据库，特别当它作为数据文件表示时更是如此。

本章的重点包括以下几点：

- 一个记录是由多个组成部分组成的，而这些组成部分称为字段。在C语言中，记录常被称为结构，其组成部分被称为成员。
- 声明一个记录分为两个步骤。首先要定义类型，为记录提供模板，之后即可声明该类型的变量。
- 给定一个记录，可以使用点运算符来选取个别字段。
- 在很多情况下，使用记录指针比记录本身更合适。从空间上讲，使用指针效率更高，而且可以通过函数修改记录的内容。

```
Welcome to ADVENTURE!
1
You are standing at the end of a road before a
small brick building.  A small stream flows out
of the building and down a gully to the south.
A road runs up a small hill to the west.
-----
south: 2
north: 8
in:    8
west:  9

2
You are in a valley in the forest beside a stream
tumbling along a rocky bed.  The stream is flowing
to the south.
-----
south: 3
down:  3
north: 1

3
At your feet all the water of the stream splashes
down a two-inch slit in the rock.  To the south,
the streambed is bare rock.
-----
north: 2
south: 4
down:  4

. . .              其他的问题              . . .
```

```
Enter name of course: advent.dat↵
Welcome to ADVENTURE!
You are standing at the end of a road before a
small brick building.  A small stream flows out
of the building and down a gully to the south.
A road runs up a small hill to the west.
south↵
You are in a valley in the forest beside a stream
tumbling along a rocky bed.  The stream is flowing
to the south.
down↵
At your feet all the water of the stream splashes
down a two-inch slit in the rock.  To the south,
the streambed is bare rock.
```

图16-3 数据文件advent.dat和相关运行示例

589 ~ 591

复习题

1. 给出记录和字段的定义。当在C语言中描述这两个概念时通常用什么术语？
2. 判断题：数组中的每个元素类型都相同。
3. 判断题：记录中每个字段的类型都必不相同。
4. 声明一个记录变量的必要步骤是什么？

```
+-------------------------------------------------+
| Scrooge and Marley Ltd.                         |
|                                                 |
| Pay to the order of: Bob Cratchit       13.00   |
|                                                 |
|                                                 |
| 15.00 gross - 2.00 tax              E. Scrooge  |
+-------------------------------------------------+
```

593

你的工作包括打印支票的外框，并对内部信息进行格式化。

2. 假设需要你为图书馆建立一个计算机处理的卡片目录系统。第一步，你需要建立一个原型，使之能够存储1000本书的如下信息：

- 书名
- 一个最多包含五位作者的名单
- 美国国会图书馆的目录号
- 至少五个主题词
- 出版商
- 出版年
- 该书是否流通

为此图书馆数据库原型设计合适的数据结构，可以存储所有必要的信息。给出你的定义后，可以按照语句：

```
libraryDB libdata;
```

进行声明，且变量libdata应该包含1000册书的信息。要记住，书的实际数量通常少于这个上界。

编写过程SearchBySubject，将图书馆数据库和一个主题词作为参数。如果某本书的主题词中包含该主题词，则SearchBySubject应显示该书的书名、第一作者的姓名，以及国会图书馆目录号码。

3. 编写函数Midpoint(p1, p2)，返回p1和p2之间线段的中点。函数的参数及结果都应该为pointT类型，其定义如下：

```
typedef struct {
    double x, y;
} pointT;
```

4. 设计接口ptgraph.h，用点代替坐标表示相对位移。例如，ptgraph.h的用户调用MovePenToPoint(pt)，而不是MovePen(x,y)，其中参数pt的类型为上题中定义的pointT类型。有些函数，如InitGraphics和DrawArc不接受坐标值作为参数，对于这类函数，在你的接口中不需重新定义。实现ptgraph.h接口，使之成为graphics.h顶部的分层抽象。

5. 可以用两个整数的商表示的数称为**有理数**（rational number）。因此1.25是一个有理数，它等于5除以4。很多数不是有理数，比如π和2的平方根。在计算时，使用有理数比使用浮点数更有优势：有理数是精确的，而浮点数不是。因此，设计一个专门处理有理数的程序是非常有用的。 594

设计一个接口rational.h，允许对有理数进行简单的操作。至少，你的接口应该可以

每一行都有某种货币的名称，后跟至少一个空格，以及一单位该货币相当的美元数。因此，示例输入文件告诉我们1英镑折合$1.96，而1德国马克折合68美分。(注意文件中有一行是美元的信息，因此它的汇率永远是1.00。这行的存在可以使美元不必作为特例处理。)

编写一个程序执行以下步骤：

a. 从exchange.dat文件中读入一个合适的内部数据结构。

b. 让用户输入两种货币的名称：将前者转换，而返回后者。

c. 让用户输入原货币的一个值。

d. 将得到的值以第二货币的形式显示。最简便的方法是，将原值转换为相应的美元数，再将此美元数转换为第二种货币数。

下面是运用exchange.dat文件运行的一个实例：

```
Convert from: mark↵
Into: yen↵
How many units of type mark? 200↵
200 mark = 17435.9 yen
```

8. 在很多应用程序中，在一个字符串与其定义之间建立起类似字典似的联系非常重要。在程序中的某一点可以对一个字符串进行定义；在之后的某一点，可以查看该字符串的定义。更正规的情况是设计一个程序，将一个值和一个关键字联系起来（Define操作），然后查找与该关键字相关的值（Lookup操作）。

图16-4中所示的接口dict.h中具有这种功能，通过调用Define(key,value)，你就可以在程序后面的某一点通过调用Lookup(key)来查找该值。运用记录数组来表示数据，实现接口dict.h。

```
/*
 * File: dict.h
 * ------------
 * This interface exports functions for defining and looking
 * up words in a dictionary. The dictionary is maintained in
 * static storage private to the implementation.
 */

#ifndef _dict_h
#define _dict_h

#include "genlib.h"

/*
 * Function: InitDictionary
 * Usage: InitDictionary();
 * ------------------------
 * This function initializes the dictionary to be empty and must be
 * called before any of the other operations are used.
 */

void InitDictionary(void);
```

图16-4 dict.h

```
Input file: formlet.txt↵
   name: Dr. Roberts↵
   topic: the misuse of computers↵
   message type: e-mail message↵

Dear Dr. Roberts:

Thank you for taking the time to express your interest in
the misuse of computers. Since being elected to Congress,
the misuse of computers has been my top priority.

Because your e-mail message means so much to me, I have
taken the time to respond personally. I hear from many
constituents every day, but your e-mail message on
the misuse of computers has moved me deeply.

Please be assured that I will do all I can to fight the
special interests that oppose us on this critical issue.

                                   Sincerely yours,

                                   Your Congressperson
```

597～599

对程序Madlibs进行扩展，存储需替换字符的字典，并在每一次出现时用恰当的值替换。

10. 图16-1所示的程序teach.c有时会产生混乱的输出结果，因为在它的设计过程中有个小问题。按照图16-2中所示的课程数据库文件，学生从程序中得到的反馈必定和下一个问题相关联。如文件cs.dat中的问题3：

```
No. Floating-point numbers have fractional parts;
integers do not.
True or false: Integers can be negative.
```

前两行为前面一个问题的答案，而最后一行是下一个问题。

这两条信息的结合必定会导致错误。例如，如果学生对于问题3的回答是不可确认的，计算机将重复这个问题。但是，问题的文本中包含的对前面问题的反馈信息就已经毫无意义了。因此，如果学生做出诸如“maybe”这样的回答，屏幕上将出现这样的显示：

```
True or false: Integers can be negative.
maybe↵
I don't understand that.
No. Floating-point numbers have fractional parts;
integers do not.
True or false: Integers can be negative.
```

考虑一下如何重新设计程序teach.c来避免这一问题，实现并测试你的新设计。 600

17.1.1 递归的简单说明

为了更好地了解什么是递归，我们举一个例子，假设你被一家大型慈善机构任命为融资协调者，你的工作是筹集1 000 000美元的资金以支付该机构的运作开销。

如果你能找到一个人愿意掏出1 000 000美元，你的工作就很简单了。然而，你很难一下子找到这么慷慨大方的人。所以，你很有可能需要募集很多小笔的捐款来凑齐1 000 000美元。如果平均每笔捐款额为100美元，你可以用另一种方法完成这项工作：找10 000个朋友说服他们每个人捐出100美元。但是你又不大可能有10 000个朋友，那你该怎么办呢？

当你面对的任务超过你个人的能力时，完成这项任务的办法就是把部分工作交给其他人做。你所在的机构应该有很多志愿者，如果你能找到10个位于全国各地的志愿者，你可以委任他们为各地的区域协调人，这样他们每个人只要筹集100 000美元就够了。

筹集100 000美元比筹集1 000 000美元简单得多，但也绝非易事。这些区域协调人又怎么解决这个问题呢？他们也可以运用相同的策略把部分筹募工作交给别人。如果他们每个人都能找到另外10个筹募志愿者，那么这些筹募志愿者每人就只需筹集10 000美元即可。这种代理的过程可以层层深入下去直到筹款人可以一次募集到所有他们需要的捐款。因为平均每笔捐款额为100美元，所以志愿者完全可能找到一个人愿意捐献这么多善款，从而无需再找更多人来代理筹款的工作了。

可以将上述筹资策略用伪代码来表示，它的结构如下所示：

```
Void CollectContributions(int n)
{
    If (n<=100){
        从一个捐赠人处收集资金
    } else {
        找十个志愿者
        让每个志愿者收集n/10美元
        把所有志愿者收集的资金相加
    }
}
```

以上伪代码中，最重要的是

```
让每个志愿者收集n/10美元
```

这一行。这个问题就是原问题的再现，只是规模较原问题小一些。这两个任务的基本特征都是一样的（募捐n美元），只是n值的大小不同。再者，由于要解决的问题实质上是一样的，你 603
可以通过调用原函数来解决它。因此，前述的伪代码可以被下列代码取代：

```
CollectContributions(n/10);
```

需要着重指出的是，如果捐款数额大于100美元，函数CollectContributions最后会调用自己。在一段程序中，函数调用自身就是递归的特征。

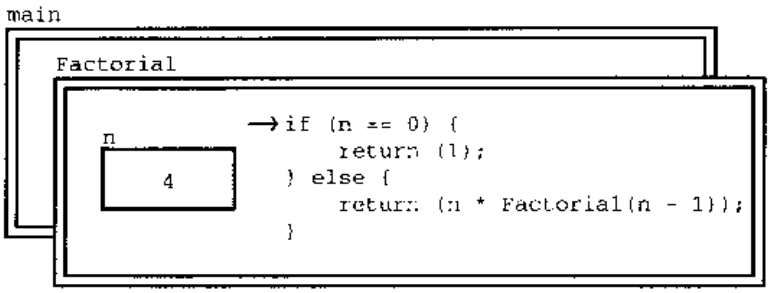
17.1.2 Factorial函数

CollectContributions这个例子之所以有用是因为它以一种浅显易懂的方式表达了递归的概念。但是，它没能告诉我们递归在实际中的用法，因为它用到的操作（如找10个志愿者，募集捐款等）都不能用C语言程序表达出来。要想从实践中了解递归的本质，我们需要找一些更适合程序设计的例子。

就像我们在第5章介绍阶乘函数时那样，用简单的数学函数来描述递归是最容易的。一个

```
fact = Factorial(4);
```

当main函数调用Factorial时，计算机创建了一个新的帧，并把实际参数值赋给形式参数n。Factorial的帧暂时取代了main的帧，如下图所示：



605

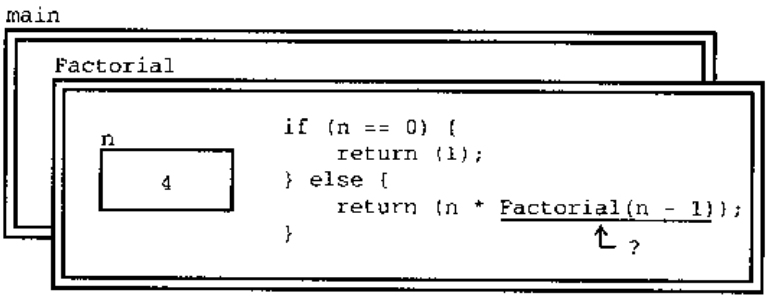
在上图中，Factorial函数体的代码显示在该帧中，这样可以使程序执行的当前位置更加清楚，我们用一个箭头表示这一点。在上图中，箭头出现在代码开始处，因为任何函数调用都是从函数体的第一句开始执行的。

现在计算机继续执行函数体，该函数从if语句开始。因为n不等于0，所以控制转而执行else子句，这时程序必须计算并返回表达式

```
n * Factorial(n-1)
```

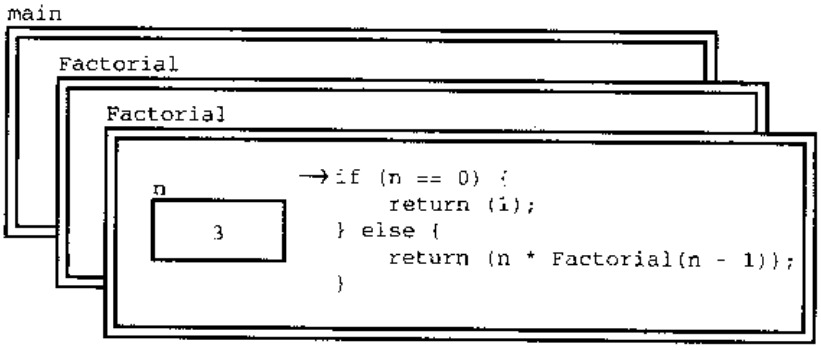
的值。

执行这个表达式需要计算Factorial(n-1)的值，这需要进行递归调用，当这个调用返回时，程序所需要做的只是将返回值乘以n。因此计算的当前状态如下图所示：



当函数Factorial(n-1)调用返回时，该返回值就替代图中下划线部分的表达式，使计算得以继续进行。

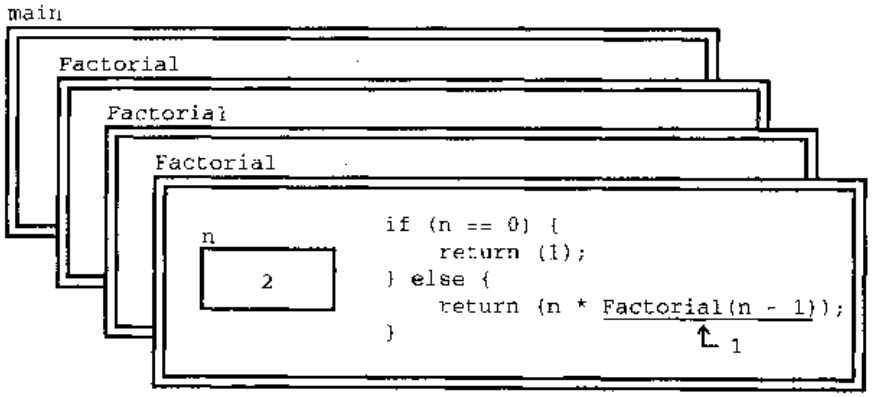
因此，下一步计算就是计算函数Factorial(n-1)的调用，这一步又是从计算实际参数表达式开始的。由于当前n值为4，所以实际参数表达式n-1的值为3。然后计算机再为Factorial创建一个新的帧，并把n-1的值赋给Factorial的形式参数。因此下一个帧如下图所示：



从这一点开始，计算就由一次次返回的递归调用所组成，用每一层Factorial的返回值计算下一层的值。例如，在现在这个帧中，函数Factorial(n-1)调用可以用1代替。因此这一层的结果可以表示为

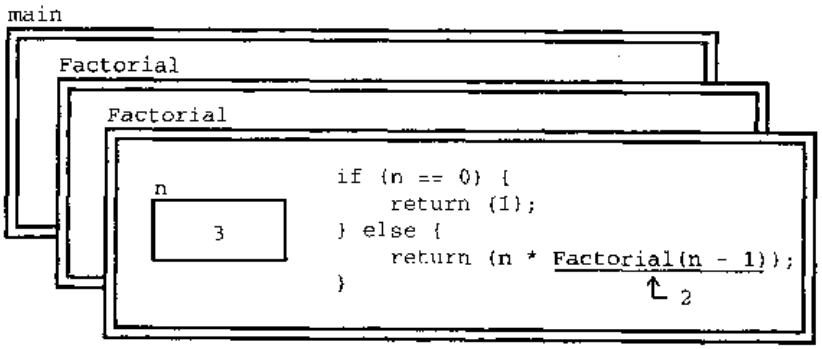
```
return (n * 1));
```

在这个帧栈中，n的值为1，所以这个调用的返回值为1，该值返回到其主调函数中，该函数可以用下图中最上层的帧表示：

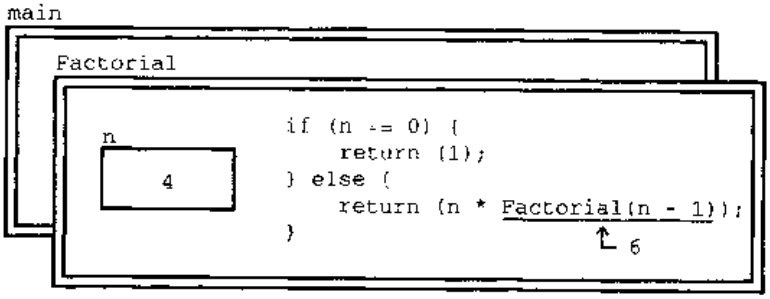


因为n现在为2，计算return语句会把2传递给前一层，如下图所示：

608



现在，程序把3×2返回到前一层，结果第一个被调用的Factorial函数的帧看上去为：



计算过程的最后一步包括计算4×6并把24返回给主程序。

17.1.3 递归信任

上一节中长长的Factorial(4)的例子告诉大家，计算机处理递归函数就像处理所有其他函数一样。当遇到一个递归函数时，至少在理论上可以模拟计算机的操作，并指出计算机要做的事。通过画出所有的帧并跟踪所有变量的值，可以重现计算机的整个操作过程并得出正确答案。然而，这样做往往会让人觉得整个过程更加复杂、更难理解。

当一个递归分解直接来自一个数学定义时，如在Factorial函数和RaiseIntToPower函数中那样，应用递归方法并不是那么困难。但是当问题本身变得越来越复杂时，情况就发生了变化，递归分解方法需要仔细思考才能找出来。下一节将解决一个更加复杂的问题，其中找到递归分解方法就比较困难。

17.1.5 排列的生成

许多文字游戏中的大部分工作是重新排列一组字母形成一个单词。例如，如果你想写一个Scrabble程序，最好能有一个工具把一组牌的所有可能的排列顺序都找出来。在文字游戏中，这样的排列通常称为**回文构词**（anagram）；在数学里，我们称之为**排列**（permutation）。

假设要写一个函数ListPermutations(s)，显示字符串s的所有排列。例如，调用

```
ListPermutations('ABC')
```

程序应该在屏幕上显示“ABC”的六种排列，如下所示：

```
ABC
ACB
BAC
BCA
CBA
CAB
```

六个排列的显示顺序并不重要，重要的是每种排列只能出现一次。

怎样编写这个ListPermutations(s)函数呢？如果只想使用循环控制结构，如for或while循环来解决问题，那么要想找到一个通用的解决方案来处理任何长度的字符串的排列问题就比较困难了。但是如果用递归思想来考虑这个问题，我们就能得到一个相对简单的解决方案。

和一般的递归程序一样，解题过程的难点在于怎样把复杂的原问题分解成简单的同类问题。在这个例子中，要为一个字符串找出所有可能的排列，你需要搞清楚为一个稍短一些的字符串生成所有排列对解题有没有帮助以及有何帮助。 611

为了对这个问题有更清楚的认识，我们来探讨一个具体的例子。假设你要为一个含5个字母的字符串“ABCDE”生成所有可能的排列。在解决方案中，你可以应用递归信任生成任何稍短的字符串的所有排列。假设递归调用是有效的，并且都已经成功返回结果了。现在，这个关键性的问题又出现了，也就是怎样利用稍短字符串的排列解决5字母字符串的排列问题。

解决这个排列问题的关键是了解以下事实：5字母字符串“ABCDE”的所有排列是由下列字符串组成的：

- 字母"A"后跟着"BCDE"的所有可能排列。
- 字母"B"后跟着"ACDE"的所有可能排列。
- 字母"C"后跟着"ABDE"的所有可能排列。
- 字母"D"后跟着"ABCE"的所有可能排列。
- 字母"E"后跟着"ABCD"的所有可能排列。

更一般化地说，要显示长度为n的字符串的所有排列，可以依次把n个字母中的每个字母置于列首，其后跟着剩下$n-1$个字母的所有排列。

这种解决方法的唯一问题在于递归子问题和原问题的形式并不完全相同，而这正违反了

的。函数PermuteWithFixedPrefix完成了所有必需的工作，但它并不是所需要的原型。当你调用PermuteWithFixedPrefix时，你需要传递一个整数和一个字符串。大多数客户只考虑排列一个字符串中的字符，这里面不包括任何整数。因此，从客户的角度考虑的话，应该像下面这样定义ListPermutations：

```
static void ListPermutations(string str)
{
    PermuteWithFixedPrefix(str, 0);
}
```

613

当用递归方案解决问题时，往往需要定义一个原型与原问题稍有不同的递归函数。就像在PermuteWithFixedPrefix中一样，新的函数经常带有一些额外的参数，但这些参数对于整个问题来说并不是必须的。这些参数记录了递归算法的进程，并且为衡量子问题是否更简单提供了一个标准。客户所用的函数调用了内部函数，给这些额外参数传递初值，就像在ListPermutations中那样。像ListPermutations那样的函数，它的目的是为一个更通用的函数提供额外参数，那么这样的函数叫做**包装**（wrapper）。

17.1.6 用递归的思想思考

学习使用递归比学习任何其他程序设计知识更要求你从整体上考虑程序设计过程。如果采用归约的方法，跟踪一个复杂递归分解的所有步骤，你最终将不得不面对大量的细节问题，而这些细节最好留给计算机去面对。前面曾介绍的Factorial(4)的整个计算过程的步骤演绎出来要用好几页纸。相比之下，整体的思想

```
Factorial (4) = 4 * Factorial (3)
```

只需要一行就可以表达清楚了。这两种递归思路在概念复杂度方面有巨大差别。从整体的观点看问题，可以降低大多数递归程序的复杂度，使你能够理解这种解题方法。然而，如果你因为对解决方案的正确性心存疑虑而选用归约的方法，大量的细节将不可避免阻碍你理解这种解决方案的复杂结构。

以正确的方法思考递归问题并非易事。要学会有效地应用递归，就需要不断地实践。对于很多学生来说，掌握这种概念需要几年的时间，但因为递归是最有用的程序设计技能之一，所以下这些功夫也是值得的。

17.2 抽象数据类型

本书介绍过，程序中的数据结构形成了一个层次结构。原子数据类型（如int、char、double和枚举类型）构成了层次中的最低层。为了表达更复杂的信息，必须把原子类型组合起来构成大一些的数据结构。而这大一些的结构又可以组合成为更大的数据结构，而且这种结合是开放式的，没有限制的。使用三种构造类型的基本元素（数组，指针，记录），可以在这个层次结构中加入新的层次。如果有一种已存在的类型，你可以定义这个类型的数组，或者把它当作一个记录的一个字段，或者声明一个指向它的指针。这三种工具是数据层次结构的粘合剂，你可以用它们建造具有任意复杂度的结构。

614

然而，当你学到更多关于程序设计的知识和技能后，你会发现特殊的数据结构有时非常有用，并且值得学习。例如，一个字符串在计算机内部表示为一个字符数组，而数组又是由一个指向数组首字母的指针来表示。在本书中，第14章介绍了字符串的基本结构。但字符串也有它的抽象行为，这些抽象行为比它的表示更重要。在第9章已经介绍过了它的抽象属性。

信放进一个队列，稍后再处理它。同样，大多数支持多用户的系统都把文件排队进行打印。当你想打印一份文档时，那个文档先进入系统的某个队列中，当打印机完成之前的那些打印请求后，就开始打印你的文档。队列的操作是一样的，但队列中存储的信息则完全不同。

关键在于使用队列软件包的客户必须决定存储在等候队列中的数据的类型。如果客户是一个电子邮件系统，那么它需要将邮件排队。如果客户是一个打印文件的程序，那么它需要排队打印请求。如果客户要的是一个模拟超市的程序，那么它必须能将以某种方法表示的超市顾客进行排队。而就队列软件包而言，它的职责是在客户提供一个数据时，它该怎样进行入队操作。当稍后执行一个出队操作时，队列软件包必须能把这个数据完整精确地送回给客户。

因此，为了使你的队列软件包尽可能通用，就应该让你的客户来控制存储在队列中的数 616
据类型。唯一的问题是你必须在接口中引用这个类型。和其他程序设计语言不同，C语言中没有一个能与任何类型相匹配的特殊类型。在C语言中，最好是用void*这个类型，它能匹配任何指针类型。如果使用void*，那么队列软件包的客户就能对任何指针类型的数据进行出队、入队操作了。因为我们总是可以为非指针型数据分配空间并使用其地址，所以在实践中，这种限制并不是什么大问题。

下一问题是怎样表示一个等候队列本身。与顾客不同，队列是实现的一个属性。队列软件包要负责在队尾使新数据入队，在队首使头一个数据出队。你的实现必须能够控制等候队列的表示，使它能进行这些操作。因此，在这个接口中，你要定义抽象队列类型queueADT，让实现部分控制基本的表示。

在C语言中，你可以在接口中定义一个类型，使它的内部表示对客户来说不可见。要做到这一点，你必须在接口中包含一个抽象类型的定义，其形式如下面的语法框所示。例如，为了把queueADT定义为一个抽象类型，接口必须包含下列定义：

```
typedef struct queueCDT *queueADT;
```

C语言的语法认为，这一行把queueADT类型定义为指向名为queueCDT的结构的指针。而queueCDT还没有被定义。因为指针总是有同样的大小，所以C编译器允许你使用结构指针，即使它还不知道结构的任何细节。一个没有被定义的结构称做**不完全类型**（incomplete type）。

语法：抽象类型的定义

```
typedef struct nameCDT * nameADT;
```

其中：

*name*ADT是抽象类型的名字。

*name*CDT是对应的具体类型的名字。

在实现中，通过写这个结构的定义来完成该类型，如下面的句法框所示。该定义包括用于标识该不完全结构的名称，它被称作**结构标记**（structure tag）。类型*name*CDT是与接口中定义的抽象类型*name*ADT相关的具体数据类型。接口的客户只能看到抽象类型。客户可以声明抽象类型的变量，对这些变量的值进行操作，就好象它们是原子类型那样。因为编译器并不知道它的基本表示是怎样的，所以客户也不能间接引用该抽象类型去看它的具体构造。另一 617
方面，这个实现也已经访问了具体类型。因为编译器现在知道了基本结构，所以实现中的函数可以间接引用抽象类型指针，从而看到具体类型中的字段。

```
/*
 * Function: FreeQueue
 * Usage: FreeQueue(queue);
 * ------------------------
 * This function frees the storage associated with queue.
 */

void FreeQueue(queueADT queue);

/*
 * Function: Enqueue
 * Usage: Enqueue(queue, obj);
 * ---------------------------
 * This function adds obj to the end of the queue.
 */

void Enqueue(queueADT queue, void *obj);

/*
 * Function: Dequeue
 * Usage: obj = Dequeue(queue);
 * ----------------------------
 * This function removes the data value at the head of the queue
 * and returns it to the client.  Dequeueing an empty queue is
 * an error.
 */

void *Dequeue(queueADT queue);

/*
 * Function: QueueLength
 * Usage: n = QueueLength(queue);
 * ------------------------------
 * This function returns the number of elements in the queue.
 */

int QueueLength(queueADT queue);

#endif
```

图 17-1 (续)

619

17.2.4 实现队列抽象

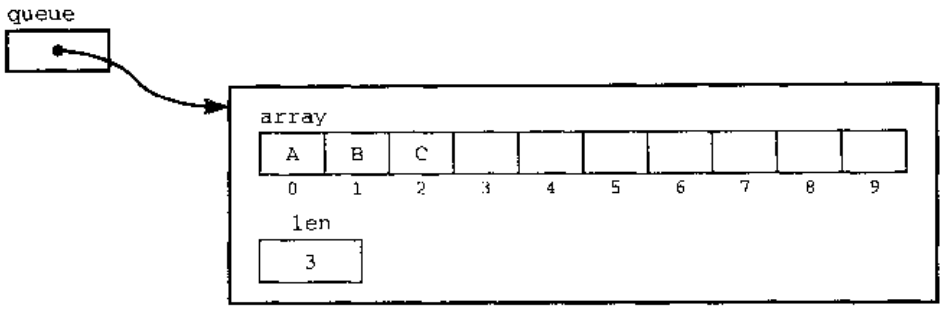
一旦完成了接口的设计，下一步就是实现队列软件包。作为实现的一部分，需要定义等候队列本身的基本表示方式。最直接的表示是一个数组和一个指示其有效长度的整数。因此，具体的类型定义如下所示：

```
struct queueCDT {
    void *array[MaxQueueSize];
    int len;
};
```

其中MaxQueueSize是一个常量，它指明了在一个队列中最多能保存的元素个数。

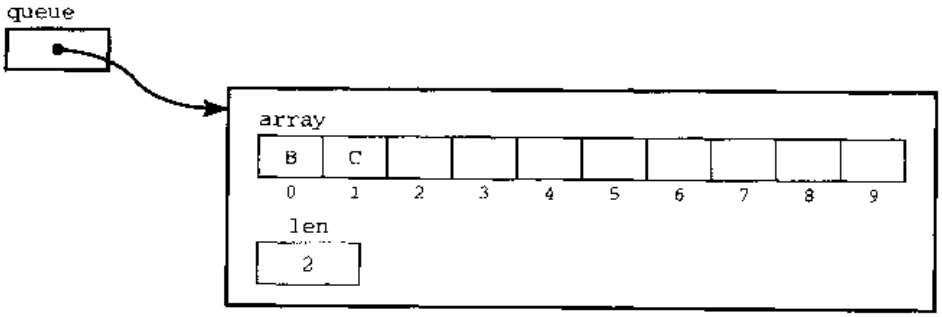
要实现函数NewQueue，要做的是为具体的结构分配空间并将队列长度初始化为0。它的实现代码如下：

```
queueADT NewQueue(void)
```



621

那Dequeue操作又如何呢？在上图所示的队列中，Dequeue操作会将A从队列中删去，并把它作为调用Dequeue的返回值返回。给定了当前的表示，唯一能把第一个元素从队列中删去的方法就是把所有其他元素都左移一位，最终的队列如下所示：



使用这种方法，Dequeue的实现如下所示：

```
void *Dequeue(queueADT queue)
{
    void *result;
    int i;

    if (queue->len == 0) Error("Dequeue of empty queue");
    result = queue->array[0];
    for (i = 1; i < queue->len; i++) {
        queue->array[i - 1] = queue->array[i];
    }
    queue->len--;
    return (result);
}
```

queue.h接口的完整实现在文件queue.c中给出，如图17-2所示。

```
/*
 * File: queue.c
 * -------------
 * This file implements the queue.h abstraction using an array.
 */

#include <stdio.h>

#include "genlib.h"
#include "queue.h"

/*
 * Constants:
```

图17-2 queue.c

```
    queue->array[queue->len++] = obj;
}

/*
 * Function: Dequeue
 * -----------------
 * This function removes and returns the data value at the
 * head of the queue.
 */

void *Dequeue(queueADT queue)
{
    void *result;
    int i;

    if (queue->len == 0) Error("Dequeue of empty queue");
    result = queue->array[0];
    for (i = 1; i < queue->len; i++) {
        queue->array[i - 1] = queue->array[i];
    }
    queue->len--;
    return (result);
}

/*
 * Function: QueueLength
 * ---------------------
 * This function returns the number of elements in the queue.
 */

int QueueLength(queueADT queue)
{
    return (queue->len);
}
```

图 17-2（续）

17.2.5 队列抽象的另一种实现方法

上一节开发的队列软件包的实现不是唯一的选择。尽管它很简单，但它有以下两个局限性：

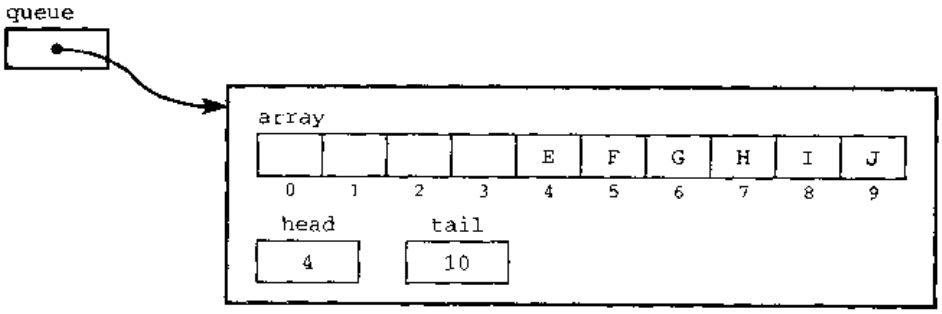
1) 调用Dequeue效率不高。每次调用Dequeue后，该实现需要把队列中剩下的项都向数组头的方向移动。对于有很多项的队列来说，这是很费时的。如果客户要把队列软件包应用于有时间要求的工作中，Dequeue的低效率就变得不可接受了。

2) 队列的最大长度是固定的，由常量MaxQueueSize指定。有些客户可能无法接受这种限制。对它们来说，只要内存允许，队列最好能动态扩展。

如果为队列结构选择一个不同的基本表示，可以同时解决这两个问题。另外，因为接口将queueADT定义为抽象类型，你可以任意改变队列的基本表示而不影响接口的客户。

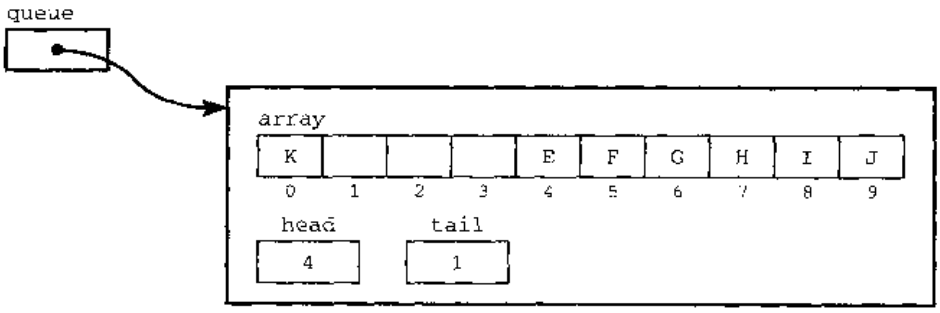
第一步，考虑怎样解决Dequeue的低效率问题。低效的根源在于函数必须对数组中的每一个元素进行移位操作，以保证第一个队列项处于下标位置0处。尽管这样简化了队列软件包的实现，但它并非是必须的做法。可以用跟踪两个下标位置的方法解决：等候队列中的第一项的下标和下一个入队的项将存入的位置的下标值。程序员称这两个位置为**队头**（head）和**队尾**（tail）。包含这两个下标的具体的类型定义如下：

G、H、I和J还在队列中。其状态如下图所示：

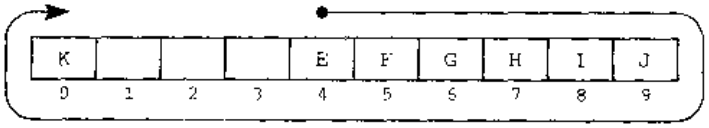


626

如果这时客户K来了会怎样呢？数组末尾没有多余的位置了。实际上，tail的下标值现在指示的元素已超过数组范围。另一方面，数组却并非处于“满”的状态。因为前四个位置空出来了。如果小心使用下标的话，你在实现队列软件包时，可以使Enqueue操作重新使用Dequeue操作释放的空间。在当前的例子中，K可以进入数组元素0处，使队列处于如下状态：

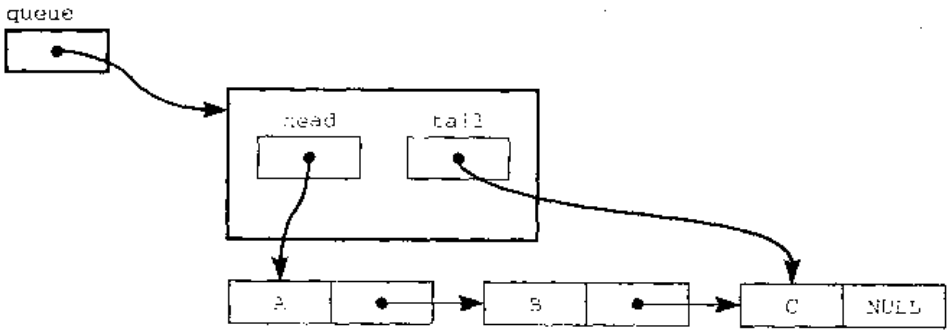


在这个设计中，数组末尾“卷回”到开头，这样数组更像一个环而非一个线性表了，如下图中的箭头所示：



因为数组开头与末尾在概念上是互相连接起来的，所以程序员称这种表示为**环缓冲区**（ring buffer）。环缓冲区常常用于实现队列，在练习7里读者有机会实现这种设计。

如果你继续深入学习计算机科学，你会发现队列常常用一种完全不同的方法实现。除了用数组保存队列外，还可以设计一种队列表示方法，其中每个队列项包含一个指向下一个队列项的指针。例如，一个队列包含A、B、C三项，如下图所示：



627

因为这个队列的每个项包含一个指向下一个项的指针，所以这种结构被称为**链表**（linked

如，考虑一下图17-2中的队列软件包的原始实现。Enqueue操作的运行时间不随问题规模的变化而改变，此处问题规模可以定义为队列中当前的项数。在计算机科学中，如果一个操作所需的时间与问题规模无关，则说该操作以**恒定时间**（constant time）运行。在*O*标记中，恒定时间用*O*(1) 表示。*O*(1) 的意思是当*N*变大时，这个程序的运行时间随 1 的改变而改变。因为1是常数，所以当*N*值增加时它并不发生变化，这也是恒定时间操作的一个显著特征。

图17-2的Dequeue操作的行为有所不同。Dequeue的实现用下列for循环把该队列中每一项向数组头方向移一步：

```
for (i = 1; i < queue->len; i++) {
    queue->array[i - 1] = queue->array[i];
}
```

如果队列包含*N*项，这个for循环就要执行*N*个周期。随着*N*的增加，for循环执行时间成正比例增加。而且，如果*N*很大，for循环的消耗远远超过循环外所有操作的消耗，因为无论*N*值多大，这些循环外的操作往往只执行一次。这样一来，在某一范围内随着*N*变大，Dequeue 629
操作的运行总时间会随*N*的增加而增加。如果一个操作的运行时间与问题规模成正比，则称此操作以**线性时间**（linear time）运行，用*O*标记表示为*O*(*N*)。

在本章前面的17.2.5节中提到的队列软件包的环缓冲区实现的优点在于，它把Dequeue操作的运行时间由线性时间降低为恒定时间，即从*O*(*N*) 到*O*(1)。如果*N*值很大，这种优势是很明显的。另一方面，使一个线性算法以恒定时间运行所节省的时间远远小于在更复杂的算法中进行改进带来的省时效果。本章剩下的内容从一个新的视角看看第12章提出的排序问题，以强调计算复杂性的问题。

17.3.3 再看选择排序

在第12章中，用于实现SortIntegerArray的算法称为*选择排序*。如果给出一个有*N*个元素的数组，选择排序算法要访问每一个数组元素的位置，并确定该位置应该存放的值。为了找到合适的值，算法必须查找剩下的数组元素并找到最小的值。这样，算法用*N*步填好第一个位置，用*N*－1步填入第二个位置，依次类推，因此，总的运行时间与

$$N+N-1+N-2+\cdots+3+2+1$$

成正比。第12章的12.2.5节介绍过，该公式可化简为：

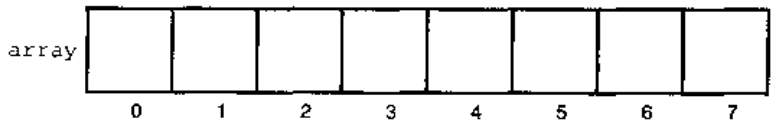
$$\frac{N^2+N}{2}$$

当使用*O*标记来估计一个算法的计算复杂性时，目的是提供一种量化手段，以便衡量当*N*变大时，*N*的变化是如何影响算法的性能。因为*O*标记并非一种精确的量化手段，所以我们最好能简化括号中的表达式，以便用最简单的形式量化算法的行为。

对圆括号里的公式应用下列步骤，可以简化*O*标记。

1) *消去公式中任何随N变大而变得无足轻重的项*。例如，在选择排序算法中，当*N*值增加时，N^2很快比*N*大很多。选择排序的运行时间因此更依赖于N^2项。因此，当使用*O*标记时，可以忽略*N*而重点关注N^2。

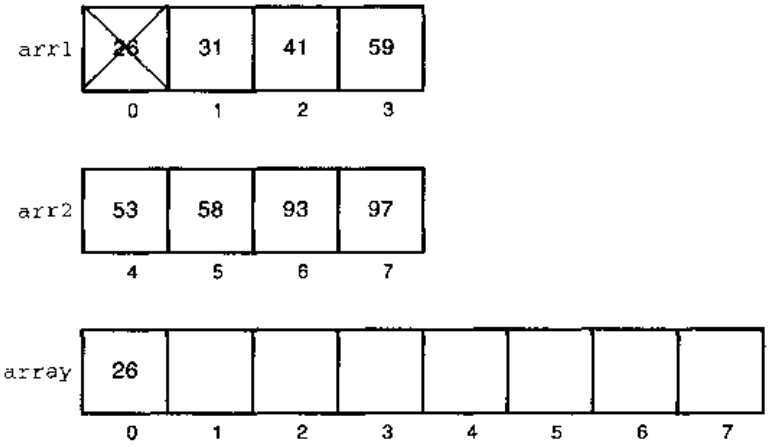
2) *消去任何常数系数*。当你计算计算复杂性时，你最关心的是对于不同的*N*值，各种算法 630
的相对运行时间。如果你用一个比率表示相对时间，那么分子和分母中的常数系数都会被消去。常数系数对于相对运行时间是没有影响的，所以可以在用*O*标记时删去常数系数。



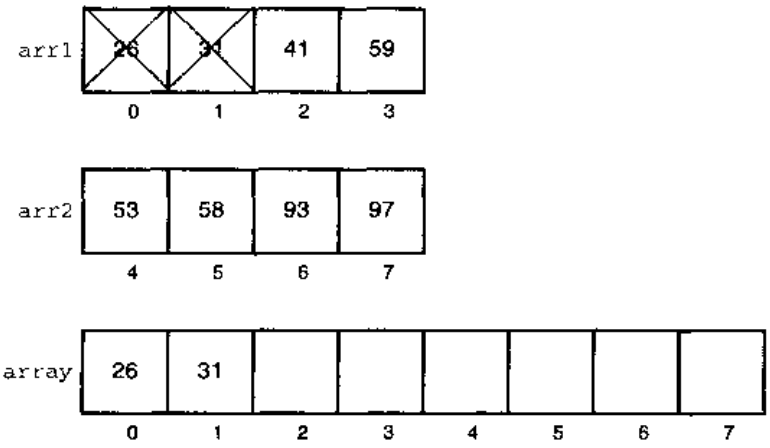
你该怎么做呢？

17.3.5 合并两个数组

把已排序的子数组重组成一个完整数组比排序本身简单多了。这种技术叫**合并**(merging)。它基于一个事实，即完整的排序的第一个元素只能是arr1或arr2的第一个元素，但要取更小的那个元素作为第一个元素。在这个例子中，新数组中的第一个元素是arr1中的26。如果 632
把这个元素放入array[0]，实际上，就是把它从arr1中划掉，结果得到以下结构：



接着，下一个元素只能是两个子数组中第一个未用过的元素。把arr1中的31和arr2中的53比较，并选择前者：



可以继续在arr1、arr2中选择较小值的过程，直到整个数组被填满为止。

17.3.6 合并排序算法

合并操作与递归分解相结合，产生了一个新的排序算法——**合并排序**(merge sort)，实现这个算法的方法很简单。函数SortIntegerArray首先判定数组的大小。如果数组没有元 633
素或只有一个元素，那么数组必然已经被排过序。因此这个条件定义了这种简单情况。然而，如果数组包含的元素多于1个，就需要执行下列步骤。

1) 把数组分成两个较小的子数组，每一个数组的大小是原数组的一半。

当调用合并排序的实现（SortIntegerArray函数）为N个数字排序时，运行时间可分成下面两部分：

1) 在当前的递归分解层次上，执行操作所需的所有时间。

2) 执行递归调用的时间。

在递归分解的最上面一层，执行非递归操作所消耗的时间与N成正比。SortIntegerArray的两个for循环一起形成N个循环周期，对Merge的调用填满了原数组的N个位置。如果你把这些操作加起来，去掉常量因子，你会发现任何一次SortIntegerArray调用的复杂性（如果不考虑其中的递归调用）需要$O(N)$次操作。

但递归操作的代价又如何呢？要排序一个大小为N的数组，必须递归地排序两个大小为$N/2$的数组。为每一个子数组排序都需要同样的时间。如果应用同样的逻辑，则很快就能够确定每一个递归调用需要的时间正比于该层上的$N/2$，加上递归调用所需时间。继续同样的过程，直到你到达一种简单情况，即子数组只有一个元素或无元素为止。

解决这个问题所需的总时间是递归分解的每一层所消耗的时间之和。总的来说，分解的结构如图17-3所示。当你沿着递归层次向下执行时，数组越来越小，但数组个数却越来越多。然而，每一层的工作量总是和N直接成正比的。因此，确定工作总量就转化为确定层数的问题。

在递归层次结构的每一层中，N值都除以2。总层数等于你在$N=1$之前将N除以2的次数。 635
用数学术语表述这个问题，就是说必须找到一个这样的k值，使得

$$N=2^k$$

解这个方程，得到k的值为：

$$k=\log_2 N$$

因为层数为$\log_2 N$，且每一层的工作量与N成正比，所以总工作量与$N\log_2 N$成正比。

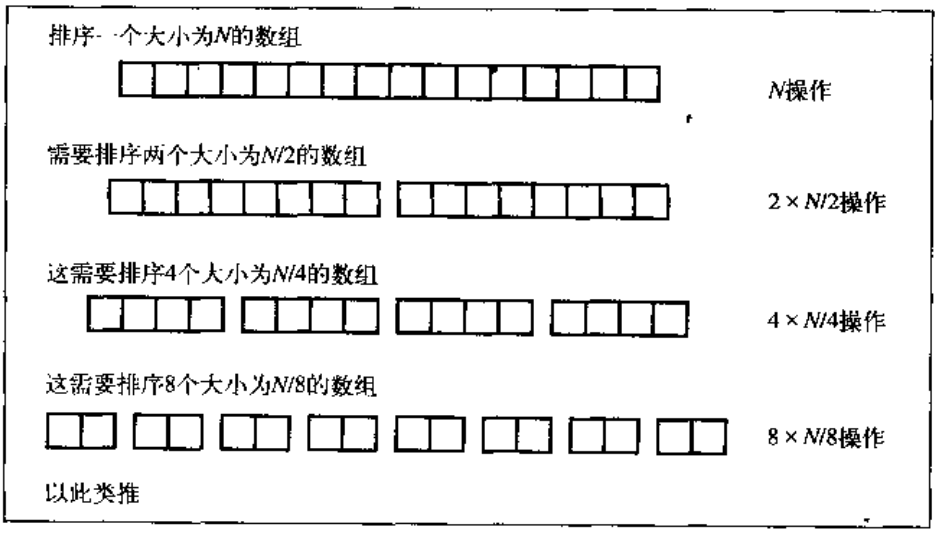


图17-3 合并排序的递归分解

在其他学科中，对数的底数通常为10（常用对数）或数学常量e（自然对数）；但在计算机科学中，总是使用**二进制对数**（binary logarithm），即以2为底数的对数。由于对数计算底数不同，而底数又是常数，因此，当我们谈到计算复杂性的问题时，我们可以像以前那样忽略对数的底数。这样合并排序的计算复杂性可以写为：

$$O(N\log N)$$

的变化而变化的。

• 降低一个算法的计算复杂性可以大大提高运行效率。

复习题

1. 定义术语*递归*。

2. 递归与逐步精化的本质区别是什么？

3. 术语*递归信任*是什么意思？

4. 典型的递归函数的函数体中第一个关键字是什么？

5. 在PermuteWithFixedPrefix的实现中，两次调用了ExchangeCharacters。第二次调用是必须的吗？说明理由。

6. 什么是包装？为什么包装在递归程序设计中如此普遍？ 638

7. 队列的基本操作是什么？

8. 假设你想设计一个抽象类型PictureADT来保存图形结构。在不知道任何基本表示的情况下，怎样在接口中定义PictureADT，以使它能和相应的具体类型相联系？

9. 一个具体类型的定义属于接口还是实现？

10. 什么是环缓冲区？将环缓冲区作为队列的基本表示主要有哪些优点？

11. 为了简化O标记的括号中的公式，可以做哪些修改？

12. 以O标记表示的选择排序的计算复杂性是什么？合并排序的复杂性又是什么？描述其区别说明了什么问题。

13. 图12-3中实现的二分查找算法的计算复杂性是什么？

程序设计练习

1. 在第4章的练习8所介绍的Fibonacci数列中，前2项为0和1，以后的每一项是该项的前2项之和。因此这个数列的开始几项为：

$$
\begin{array}{llll}
F_0 & = & 0 & \\
F_1 & = & 1 & \\
F_2 & = & 1 & (F_0 + F_1) \\
F_3 & = & 2 & (F_1 + F_2) \\
F_4 & = & 3 & (F_2 + F_3) \\
F_5 & = & 5 & (F_3 + F_4) \\
F_6 & = & 8 & (F_4 + F_5)
\end{array}
$$

用同样的方式生成后续项。写出函数Fib(n)的递归实现，使它返回第n个Fibonacci数。你的实现必须只依赖于数列中项之间的关系，而不能用任何迭代结构，如for或while。

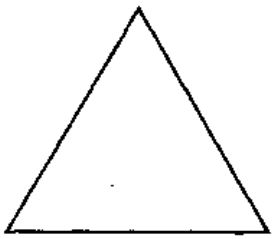
2. 第5章介绍的数学中的组合函数C(n, k) 常常用阶乘来定义，其定义如下：

$$C(n,k)=\frac{n!}{k!\times(n-k)!}$$ 639

C(n, k) 的值也可以排列成一个三角形，其中n值由上至下增大，k值从左向右增大。这个结构叫Pascal三角形，如右图所示。第5章的练习12指出，Pascal三角形有一个有趣的特征，除了左、右两边的项都为1以外，它的每一项都为其上两项之和。例如，考虑下页的图所示的Pascal三角形中被圆圈住的项：

C(0, 0)
C(1, 0) C(1, 1)
C(2, 0) C(2, 1) C(2, 2)
C(3, 0) C(3, 1) C(3, 2) C(3, 3)
C(4, 0) C(4, 1) C(4, 2) C(4, 3) C(4, 4)

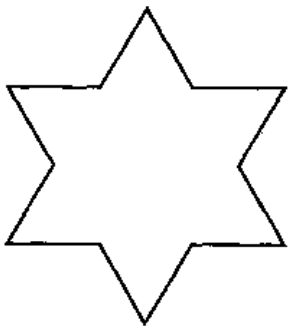
最早的一种分形之一是*koch*雪花，这是以它的发明者Helge von koch命名。Koch雪花以一个等边三角形开始：



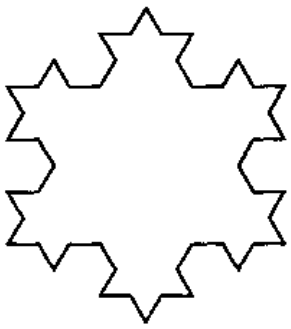
它被称为0阶koch分形。之后这个图形被逐步修改生成了更高阶的分形。在每一步修改中，三角形的每一条边被分为三段，中间一段线段被一个三角形的凸形取代。这样，如果你把三角形的每一条边都用下图做改变：



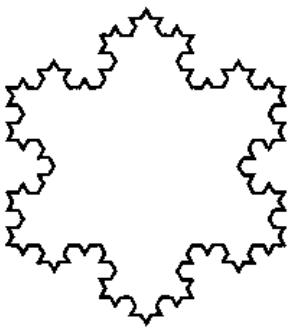
你将得到1阶koch分形：



如果你把这个图形中的每一条边再用一条有凸起三角形的线段取代，你将得到2阶koch分形： 642



再次用同样的方法替换图中的每一条边，将得到下图所示的3阶koch分形，看起来像一片雪花：



9. 在第16章的练习8中，要求你实现一个字典模块，能够存储键值和它们的相关定义。然而，按照该练习给定的设计思路，整个程序中只能有一个字典。在某些应用中，我们可能需要多个字典用于不同的目的。

重新设计dict.h接口，以实现下列变化：

a. 定义一个抽象数据类型dictionaryADT，使它能够存储所有用来表示一个字典的信息。

b. 用函数NewDictionary替代InitDictionary函数，NewDictionary能分配新空间，并返回一个新的dictionaryADT。

c. 输出函数FreeDictionary，它能释放一个已存在的字典的空间。

d. 在Define和Lookup的原型中，增加一个类型为dictionaryADT的参数，使它们知道目前正在使用的是哪个字典。

10. 如表17-2所示，选择排序对于小数组来说比合并排序快，尽管当数组变大时，情况相反。如果你想设计一个函数SortIntegerArray，使之适应于各种大小的数组，那么可以把两种策略结合起来，对大数组用合并排序，对小数组使用选择排序。用这种方法重新实现函数SortIntegerArray，那么你该如何选择区分大小数组的那个分界点呢？ 645

附　录

附录A　C语言的语法和结构总结

本附录提供了本书中出现的关于ANSI C的语法规则和结构的快速参考。尽管本附录确实包括了大多数重要的特性，但并不是C语言的参考指南。要找到关于某个特殊主题的更多信息，请参考相应的章节。

A.1　编译过程概述

一个C语言程序由一个或多个源文件组成，每一个源文件又由可读的文本组成，这些文本是用称为ANSI C的高级程序设计语言写成的。C编译器把每一个源文件翻译成相应的目标文件，目标文件中包含着在一个特定的计算机系统上执行程序所需要的指令。然后，将这些目标文件以及其他用于实现各种常用操作的称为库的目标文件链接起来创建一个可执行程序。

编译一个源文件的过程包括下列四个阶段：

1) *预处理*。在预处理时，C编译器用一组已有的操作变换整个源文件的文本，这些操作包括读入库接口和以符号常量的定义替代符号常量名。

2) *词法分析*。在词法分析阶段，编译器收集经过预处理的源文件中的字符以形成有意义的单元，这种单元叫*记号*。

3) *语法分析*。这个阶段包括检验源文件中的记号是否符合C语言的语法规则，或者句法规则。此外，语法分析阶段还负责解释程序的意义，即语义。

4) *生成代码*。编译的最后一个阶段包括把经过语法分析的程序表示翻译成执行程序所必须的机器指令。尽管这一阶段对生成目标文件是必须的，但它对源文件并没有直接影响，因此与程序员没什么关系。

前两个阶段，即预处理和词法分析，将在A.2节和A.3节中分别讨论。A.4～A.8节将着重讨论语法分析阶段并回顾C的语法结构。A.9节回顾本书用到的ANSI库函数。

A.2　C语言预处理器

预处理器完成C语言中的某些功能，本书仅详细讨论了其中的两个功能，即定义常量和包含接口文件。它们将在下面两个小节中讨论。

646～648

A.2.1　#define规范

形如

```
#define name definition
```

的源程序行定义符号*name*等价于*definition*。在源文件剩下的部分中，预处理器寻找任何作为一个完整的记号出现的*name*，用*definition*中的字符替换*name*。例如，在遇到语句

```
#define Pi 3.14159265
```

用于分隔源文件中的其他记号，否则就被忽略掉。因此空格可以帮助人们理解代码的结构。习惯上，对程序的每一行的缩排都是为了使控制结构更为清晰。同样的，空行用来分隔程序的不同部分。

A.3.2 注释

注释由字符/*和*/之间的文本组成。注释文本可以跨越很多行，但一个注释不能嵌套在另一个注释中。注释文本是为了方便程序的读者，但编译器会忽略注释。 650

A.3.3 标识符

程序中的变量、函数、类型等的名字被统称为标识符。在C语言中，构造标识符的规则为：

1) 标识符必须以字母或下划线开头。

2) 标识符中所有其他字符必须为字母、数字或下划线，不能有空格或其他特殊字符。

3) 标识符不能使用下列关键字：

```
auto        double      int         struct
break       else        long        switch
case        enum        register    typedef
char        extern      return      union
const       float       short       unsigned
continue    for         signed      void
default     goto        sizeof      volatile
do          if          static      while
```

标识符中的大写和小写字母是不同的。例如，名字ABC和名字abc是不同的。标识符可以为任意长度，但C编辑器在判断两个名字是否相同时不考虑第31个字符以后的字符。某些实现可能会对模块间共用的标识符附加一些额外的限制。

A.3.4 常量

程序可以包含下列通用类型的常量：整型、浮点型、字符型和字符串。这些类型将在下面讨论。每一种类型常量的构成规则如下：

- 整型。整型常量一般写成一个数字字符串，以表示一个十进制的数。如果这个数以0开头，则编辑器把这个数当作八进制整数，因此，常量040被当作一个八进制数，代表的十进制数为32。如果你在一个数字常量前加前缀0x，那么编译器将把它解释为十六进制数。因此，常量0xFF对应的十进制数为255。你可以在数字串末尾加字母L，明确表示该数字是一个长整型常量。因此常量0L等于0，但这个值是长整型的。同样，如果你把字母U作为后缀，这个常量被当作是无符号常量。
- 浮点型。C语言中的浮点常量有小数点。因此，如果程序中出现2.0，那么这个数在机器内部表示为一个浮点型的值；如果程序员写成2，那么这个值将是一个整数。浮点型常量也可以用科学计数法写成*x.xxxx*E±*yy*的形式，表示*x.xxxx*乘以10的*yy*次方。 651
- 字符型。在C语言中你可以用单引号把字符括起来表示字符型常量。该字符的值由编码模式（通常为ASCII）给出，该模式将每一个字符映射成一个数值。除了标准字符外，C语言还支持下列表示特殊字符的转义序列：
 - '\a' 震铃字符（终端的嘟嘟声）
 - '\b' 退格
 - '\f' 换页（开始一个新的页面）
 - '\n' 换行
 - '\r' 回车（返回至本行首位）

A.5 语句

本节回顾C语言中可用的语句形式。

A.5.1 简单语句

C语言中最常见的语句是简单语句，它由一个表达式后面跟一个分号组成：

```
表达式;
```

A.5.2 空语句

一个分号本身组成了一个没有任何作用的语句。这种语句形式叫做*空语句*，常用在一个控制结构体中，在这种控制结构中，重要工作都在头语句中完成了。

A.5.3 语句块

可以用花括号把几条语句括在一起，形成一个*语句块*：

```
{
    …可选的声明…
    …语句…
}
```

653

然后你可以将所产生的语句块作为任何控制语句中的一个语句。如范例所示，在任何块中的语句之前可以有变量声明。在本书中，变量声明只用于定义函数体的块内。声明将在本附录后面A.6节中深入讨论。

A.5.4 if语句

if语句用来表示简单的条件执行，并以下面两种形式出现

```
if (表达式) 语句
```

和

```
if (表达式) 语句1 else 语句2
```

A.5.5 switch语句

switch语句在一个表达式的值指定了几种相互独立的情况中的一种时使用。该语句的形式为：

```
switch (e) {
  case c1:
    语句
    break;
  case c2:
    语句
    break;
…更多的case子句…
  default:
    语句
    break;
}
```

可以有任意多个case子句，但每一个case子句所用的值必须是某种标量类型的常量。default子句是可选的，但推荐使用这个子句，如果没有default子句，并且switch行中的值不能与任何一个case子句的值相匹配，则程序会从switch语句退出，不执行任何动作。

出现在switch语句中break语句并非C语言的语法所必须的。当一个break语句被执行时，控制立刻转移到整个switch语句后面的语句。如果break语句（或其他任何改变控制

言程序设计所必须的。更重要的是，这些控制形式很容易因为滥用而造成程序结构的复杂化。下面将分别描述这三种语句，这样你在实际中遇上它们时就能理解其作用了。

do语句类似于while，只是测试是在循环末尾而非开头进行。do语句的一般形式为：

```
do{
    语句
}while (条件表达式);
```

尽管人们通常更喜欢用while，但这个语句在有些场合很有用。而用do语句的程序更容易出错，因为do循环体至少运行一次，哪怕条件表达式一开始就为FALSE也是这样。

continue语句的形式为：

```
continue;
```

它使程序立即开始执行离循环最近的下一个循环周期。尽管这非常方便，但它也使程序的可读性降低，尤其是用于复杂循环结构时更是如此。

C语言可用的最低级的控制是goto语句，其一般形式为：

```
goto label;
```

标识符*label*指定当前函数中的某一条语句，该语句用标识符*label*及一个冒号作为标记，如下所示：

```
label: 语句
```

goto语句使控制直接转到被*label*标记的语句。而goto的不规则的使用会破坏程序的结构，使它变得极其难懂，难以维护。本书中没有使用goto语句，建议你也不要使用它。 656

A.6 函数

C程序中的所有可执行语句都出现在一个函数体内。当计算机执行一个程序时，它从调用main函数开始。而main函数又调用其他函数，这些函数中分别解决一部分问题；这些函数可能又调用别的可以进一步细化问题的函数。把一个问题分解成更简单的部分的过程叫做逐步精化。

A.6.1 函数定义

函数定义有如下的句法形式：

```
result-type  name(argument-specifiers)
{
    ...optional declarations...
    ...body...
}
```

在这个范例中，*result-type*是函数返回值的类型，*name*是函数名，*argument-specifiers*是一个形式参数声明表，形式参数之间以逗号隔开。*optional declarations*部分包含函数中使用的局部变量的声明。而*body*则由实现函数所需的语句组成。

A.6.2 函数原型

在C程序中使用某一函数前，应该先通过指定它的原型[⊖]来声明它。原型和函数定义的形式完全相同，只是原型的后面不是整个函数体而是一个分号。在原型中可以不写形参变量名，尽管提供参数名通常对读者是有帮助的。

⊖ 为了与早期的C语言版本兼容，如果没有明确提供函数原型，编译器会假设它具有某些特性。在你自己的代码中，应该遵循本书的实例，为每个函数定义原型。

整个集合。如果该复合类型包含多层数据结构，那么可以使用更多的花括号来指定内部结构。

A.8 数据类型

C语言中的每一个值都与一个数据类型相关联，数据类型是用它的值域和一组操作来刻画的。这些数据类型分为两类：不再包含内部组成部分的原子类型以及包含其他类型值的复合类型。

A.8.1 原子类型

C语言中定义了下列原子类型：

- **整型** 基本的整数类型是int，它对应于一个有符号整数。表示整数的位数随机器的不同而不同，从而影响到int型的最大值。所有兼容ANSI的编译器必须允许int型至少有16位，也就是说它们的数值范围至少必须为－32 768至32 767。为了使编译器能够为一个整数分配更多或更少的空间，整型int可以用long和short加以修饰。此外，整型int、short和long可以单独被signed或unsighed修饰，表示是否允许负值。当使用修饰符时，可以省略类型名int。
- **浮点数** C语言定义了三种浮点类型：float、double和long double。这几种类型的精度和所需的存储空间取决于编译器和基本的硬件。对大部分应用来说，用double型就够了。
- **字符型** char型用来表示一个单独的字符。因为char型的值被定义为只需要一个字节的存储空间，因此字符型是表达数据值大小的基本单元。
- **枚举类型** 除了内置的原子类型外，C语言还允许你定义枚举类型，它们的值是有限个已命名的常量的集合。在本书中，用来引入一个新的枚举类型的标准句法是：

```
typedef enum {list of elements} type name;
```

659

接口genlib.h用枚举类型定义bool类型，定义方法如下：

```
typedef enum {FALSE, TRUE} bool;
```

每个枚举常量被赋予一个从0开始的整数。这样，在bool类型中，FALSE的内部值为0，TRUE的内部值为1。

A.8.2 数组

数组是一组存储在连续的存储单元中的有序的、同质的值。声明一个数组的标准句法是：

```
element type name[size];
```

在这个范例中，*element type*说明存储于数组的每个元素中的值的类型，*name*是数组变量的名字，*size*表示分配的元素个数。如果你不能确定要在数组中存放多少个元素，通常的方法是用一个大于你的应用所需的数作为*size*的值，然后只使用所分配空间的一部分。如果这样做的话，还需要声明一个整数来记录在某一时刻数组中有多少个有效元素。相对于分配长度，这个整数指出了数组的有效长度。

在一个数组中，每一个元素都用一个数字下标来标识。在C语言中，数组下标值总是从0开始。要从数组中选择一个特定元素，需要用一个选择表达式，如array[i]，其中数组名后跟一个括在方括号里的表达式，该表达式指出所要找的元素下标值。

A.8.3 指针

指针是一个数值的地址。要声明一个变量指向特定目标类型的值，C语言用下列声明形式：

```
目标类型 *名字;
```

A.9 ANSI库

本节描述了本书中用到的ANSI库函数，按接口来排列。

A.9.1 标准I/O库（stdio.h）

标准I/O库提供了一套操纵文件和执行输入/输出操作的函数，stdio.h接口通常被包含在每一个.C文件中。

• FILE *fopen(string filename, string mode);

该函数打开一个文件，并返回一个指向该文件的指针。文件的名字由字符串filename给出，字符串mode指定数据传输的类型，通常为下列各值之一：

- "r" 以读方式打开文件。文件必须已经存在。
- "w" 以写方式打开文件。如果文件存在，所有原来内容都被删除。
- "a" 以添加方式打开文件。如果文件存在，新输出的数据出现在文件末尾。

如果出现错误（如以读方式打开文件，但该文件不存在），则fopen返回NULL。

• void fclose(FILE *fp);

这个函数将fopen生成的文件指针作为参数，并关闭它。

• int getc(FILE *infile);
 int getchar(void);

getc函数读文件infile，并返回它的下一个字符。返回的值是一个整数（而非字符），因此结果类型可以包含一个特殊值EOF，用来表示文件结束。getchar函数功能相同，只是它总是从标准输入文件中读取它的输入。

• int ungetc(char ch, FILE *infile);

这个函数把字符ch推入infile指定的文件流，这样下一个从文件读取字符的函数就会把ch再次读出来。每次只有一个字符能被推回文件流。 662

• int putc(char ch, FILE *outfile);
 int putchar(char ch);

putc函数把一个字符ch写到outfile指定的输出文件中。putchar函数与之类似，只是它总是把字符写到标准输出文件。

• int fgets(char buffer[], int max, FILE *infile);

这个函数从输入文件中读入一行文本并把它们存入buffer中，buffer必须是一个由客户分配的字符数组。用这个函数最多可以读入max个字符，允许客户自己控制以防止缓冲区溢出。换行字符作为行结束标志也存储在缓冲区中。

• int fputs(string s, FILE *outfile);

这个函数把字符串s写入文件outfile中。

• void printf(string msg,. . .);
 void fprintf(FILE *outfile, string msg,. . .);
 void sprintf(char buffer[], string msg,. . .);

printf函数在标准输出设备上显示消息msg，该标准输出设备一般为终端屏幕，fprintf和sprintf函数类似于printf，只是它们输出到outfile或buffer。在这三种形式的任意一种中，msg中的字符被复制到输出流，如果在msg中出现一个百分号，它表示在该位置要用某个值替换。这个值为下一个未用的参数，这样，替代值在msg中出现的顺序将与在参数列表中出

此函数返回小于或等于x的最大整数的浮点表示。

• double ceil(double x);

此函数返回大于或等于x的最小整数的浮点型表示。

• double fmod(double x, double y);

此函数返回x/y的余数的浮点数。

• double sqrt(double x);

此函数返回x的平方根。

• double pow(double x, double y);

此函数返回x^y。

• double exp(double x);

此函数返回e^x。

• double log(double x);

此函数返回x的自然对数。

• double sin(double theta);

此函数返回角度为theta的正弦函数值，角度以弧度表示。

• double cos(double theta);

此函数返回角度为theta的余弦函数值，角度以弧度表示。

• double atan(double x);

此函数返回值x的反正切函数值，结果是$-\pi/2$到$+\pi/2$之间的角度，以弧度表示。

• int atan2(double y, double x);

此函数返回x轴与从原点到点（x, y）之间的直线所形成的角度。与其他三角函数一样，该角度以弧度表示。 665

A.9.4 字符类型库（ctype.h）

该接口输出一组极有效率的函数来分类不同的字符。[⊖]

```
• bool isalpha(char ch);
  bool isupper(char ch);
  bool islower(char ch);
  bool isdigit(char ch);
  bool isalnum(char ch);
  bool ispunct(char ch);
  bool isspace(char ch);
```

如果字符ch在指定的类别内，这些函数返回TRUE，如下所示：

```
isalpha  字母字符
isupper  大写字母字符
islower  小写字母字符
isdigit  十进制数字
isalnum  字母字符和数字的并集
```

⊖ ctype.h接口中的大部分函数定义为使用int类型，这样这些函数可以调用EOF常量。这里的原型使用char是为了增强可读性。

附录B 库源代码

此附录包含本书所用到的扩展库的接口和具体实现。这些库被设计成可移植的，可以在任何遵循ANSI标准的编译器上使用。

这些库的源代码以及本书中用到的所有程序实例的代码，可以登录www.hzbook.com来申请。

如果你有其他问题，还请写信到下列地址：

The Art & Science of C
Computer Science:Higher Ed Editorial
Addison-Wesley Publishing Company
One Jacob Way
Reading, MA 01867
U.S.A.

668
~
670

```
/*
 * File: genlib.h
 * --------------
 * This file contains several definitions that form the
 * core of a general-purpose ANSI C library developed by Eric
 * Roberts.  The goal of this library is to provide a basic
 * set of tools and conventions that increase the readability
 * of C programs, particularly as they are used in a teaching
 * environment.
 *
 * The basic definitions provided by genlib.h are:
 *
 *    1.  Declarations for several new "primitive" types
 *        (most importantly bool and string) that are
 *        used throughout the other libraries and
 *        applications as fundamental types.
 *
 *    2.  A new set of functions for memory allocation.
 *
 *    3.  A function for error handling.
 */

#ifndef _genlib_h
#define _genlib_h

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

/* Section 1 -- Define new "primitive" types */
```

图B-1 genlib.h

```
/* Section 2 -- Memory allocation */

/*
 * General notes:
 * --------------
 * These functions provide a common interface for memory
 * allocation.  All functions in the library that allocate
 * memory do so using GetBlock and FreeBlock.  Even though
 * the ANSI standard defines malloc and free for the same
 * purpose, using GetBlock and FreeBlock provides greater
 * compatibility with non-ANSI implementations, automatic
 * out-of-memory error detection, and the possibility of
 * substituting a garbage-collecting allocator.
 */

/*
 * Function: GetBlock
 * Usage: ptr = (type) GetBlock(nbytes);
 * -------------------------------------
 * GetBlock allocates a block of memory of the given size.  If
 * no memory is available, GetBlock generates an error.
 */

void *GetBlock(size_t nbytes);

/*
 * Function: FreeBlock
 * Usage: FreeBlock(ptr);
 * ----------------------
 * FreeBlock frees the memory associated with ptr, which must
 * have been allocated using GetBlock, New, or NewArray.
 */

void FreeBlock(void *ptr);

/*
 * Macro: New
 * Usage: p = New(pointer-type);
 * -----------------------------
 * The New pseudofunction allocates enough space to hold an
 * object of the type to which pointer-type points and returns
 * a pointer to the newly allocated pointer.  Note that
 * "New" is different from the "new" operator used in C++;
 * the former takes a pointer type and the latter takes the
 * target type.
 */

#define New(type) ((type) GetBlock(sizeof *((type) NULL)))

/*
 * Macro: NewArray
 * Usage: p = NewArray(n, element-type);
 * -------------------------------------
```

673

图 B-1 (续)

```
void *GetBlock(size_t nbytes)
{
    void *result;

    result = malloc(nbytes);
    if (result == NULL) Error("No memory available");
    return (result);
}

void FreeBlock(void *ptr)
{
    free(ptr);
}

/* Section 3 -- Basic error handling */

void Error(string msg, ...)
{
    va_list args;

    va_start(args, msg);
    fprintf(stderr, "Error: ");
    vfprintf(stderr, msg, args);
    fprintf(stderr, "\n");
    va_end(args);
    exit(ErrorExitStatus);
}
```

图 B-2 (续)

675

```
/*
 * File: simpio.h
 * --------------
 * This interface provides access to a simple package of
 * functions that simplify the reading of input data.
 */

#ifndef _simpio_h
#define _simpio_h

#include "genlib.h"

/*
 * Function: GetInteger
 * Usage: i = GetInteger();
 * ------------------------
 * GetInteger reads a line of text from standard input and scans
 * it as an integer.  The integer value is returned.  If an
 * integer cannot be scanned or if more characters follow the
 * number, the user is given a chance to retry.
 */

int GetInteger(void);

/*
 * Function: GetLong
```

图B-3 simpio.h

```
#include <stdio.h>
#include <string.h>
#include "genlib.h"
#include "strlib.h"
#include "simpio.h"

/*
 * Constants:
 * ----------
 * InitialBufferSize -- Initial buffer size for ReadLine
 */

#define InitialBufferSize 120

/* Exported entries */

/*
 * Functions: GetInteger, GetLong, GetReal
 * ---------------------------------------
 * These functions first read a line and then call sscanf to
 * translate the number.  Reading an entire line is essential to
 * good error recovery, because the characters after the point of
 * error would otherwise remain in the input buffer and confuse
 * subsequent input operations.  The sscanf line allows white space
 * before and after the number but no other extraneous characters.
 */

int GetInteger(void)
{
    string line;
    int value;
    char termch;

    while (TRUE) {
        line = GetLine();
        switch (sscanf(line, " %d %c", &value, &termch)) {
          case 1:
            FreeBlock(line);
            return (value);
          case 2:
            printf("Unexpected character: '%c'\n", termch);
            break;
          default:
            printf("Please enter an integer\n");
            break;
        }
        FreeBlock(line);
        printf("Retry: ");
    }
}

long GetLong(void)
{
    string line;
    long value;
    char termch;
```

677

678

图 B-4 (续)

```
 * ------------------
 * This function operates by reading characters from the file
 * into a dynamically allocated buffer.  If the buffer becomes
 * full before the end of the line is reached, a new buffer
 * twice the size of the previous one is allocated.
 */

string ReadLine(FILE *infile)
{
    string line, nline;
    int n, ch, size;

    n = 0;
    size = InitialBufferSize;
    line = GetBlock(size + 1);
    while ((ch = getc(infile)) != '\n' && ch != EOF) {
        if (n == size) {
            size *= 2;
            nline = (string) GetBlock(size + 1);
            strncpy(nline, line, n);
            FreeBlock(line);
            line = nline;
        }
        line[n++] = ch;
    }
    if (n == 0 && ch == EOF) {
        FreeBlock(line);
        return (NULL);
    }
    line[n] = '\0';
    nline = (string) GetBlock(n + 1);
    strcpy(nline, line);
    FreeBlock(line);
    return (nline);
}
```

图 B-4 (续)

680

```
/*
 * File: strlib.h
 * --------------
 * The strlib.h file defines the interface for a simple
 * string library.  In the context of this package, strings
 * are considered to be an abstract data type, which means
 * that the client relies only on the operations defined for
 * the type and not on the underlying representation.
 */

/*
 * Cautionary note:
 * ----------------
 * Although this interface provides an extremely convenient
 * abstraction for working with strings, it is not appropriate
 * for all applications.  In this interface, the functions that
 * return string values (such as Concat and SubString) do so
```

图B-5 strlib.h

```
 * 2. If p2 is greater than the index of the last string
 *    position, which is StringLength(s) - 1, then p2 is
 *    set equal to StringLength(s) - 1.
 * 3. If p2 < p1, SubString returns the empty string.
 */

string SubString(string s, int p1, int p2);

/*
 * Function: CharToString
 * Usage: s = CharToString(ch);
 * ----------------------------
 * This function takes a single character and returns a
 * one-character string consisting of that character.  The
 * CharToString function is useful, for example, if you
 * need to concatenate a string and a character.  Since
 * Concat requires two strings, you must first convert
 * the character into a string.
 */

string CharToString(char ch);

/*
 * Function: StringLength
 * Usage: len = StringLength(s);
 * -----------------------------
 * This function returns the length of s.
 */

int StringLength(string s);

/*
 * Function: CopyString
 * Usage: newstr = CopyString(s);
 * ------------------------------
 * CopyString copies the string s into dynamically allocated
 * storage and returns the new string.  This function is not
 * ordinarily required if this package is used on its own,
 * but is often necessary when you are working with more than
 * one string package.
 */

string CopyString(string s);

/* Section 2 -- String comparison functions */

/*
 * Function: StringEqual
 * Usage: if (StringEqual(s1, s2)) ...
 * -----------------------------------
 * This function returns TRUE if the strings s1 and s2 are
 * equal.  For the strings to be considered equal, every
 * character in one string must precisely match the
 * corresponding character in the other.  Uppercase and
 * lowercase characters are considered to be different.
 */
```

682

图 B-5 (续)

```
 * ---------------------------------
 * This function returns a new string with all
 * alphabetic characters converted to upper case.
 */

string ConvertToUpperCase(string s);

/* Section 5 -- Functions for converting numbers to strings */

/*
 * Function: IntegerToString
 * Usage: s = IntegerToString(n);
 * ------------------------------
 * This function converts an integer into the corresponding
 * string of digits.  For example, IntegerToString(123)
 * returns "123" as a string.
 */

string IntegerToString(int n);

/*
 * Function: StringToInteger
 * Usage: n = StringToInteger(s);
 * ------------------------------
 * This function converts a string of digits into an integer.
 * If the string is not a legal integer or contains extraneous
 * characters, StringToInteger signals an error condition.
 */

int StringToInteger(string s);

/*
 * Function: RealToString
 * Usage: s = RealToString(d);
 * ---------------------------
 * This function converts a floating-point number into the
 * corresponding string form.  For example, calling
 * RealToString(23.45) returns "23.45".  The conversion is
 * the same as that used for "%G" format in printf.
 */

string RealToString(double d);

/*
 * Function: StringToReal
 * Usage: d = StringToReal(s);
 * ---------------------------
 * This function converts a string representing a real number
 * into its corresponding value.  If the string is not a
 * legal floating-point number or if it contains extraneous
 * characters, StringToReal signals an error condition.
 */

double StringToReal(string s);

#endif
```

图 B-5 (续)

```
{
    int len;

    if (s == NULL) Error("NULL string passed to IthChar");
    len = strlen(s);
    if (i < 0 || i > len) {
        Error("Index outside of string range in IthChar");
    }
    return (s[i]);
}

string SubString(string s, int p1, int p2)
{
    int len;
    string result;

    if (s == NULL) Error("NULL string passed to SubString");
    len = strlen(s);
    if (p1 < 0) p1 = 0;
    if (p2 >= len) p2 = len - 1;
    len = p2 - p1 + 1;
    if (len < 0) len = 0;
    result = CreateString(len);
    strncpy(result, s + p1, len);
    result[len] = '\0';
    return (result);
}

string CharToString(char ch)
{
    string result;

    result = CreateString(1);
    result[0] = ch;
    result[1] = '\0';
    return (result);
}

int StringLength(string s)
{
    if (s == NULL) Error("NULL string passed to StringLength");
    return (strlen(s));
}

string CopyString(string s)
{
    string newstr;

    if (s == NULL) Error("NULL string passed to CopyString");
    newstr = CreateString(strlen(s));
    strcpy(newstr, s);
    return (newstr);
}

/* Section 2 -- String comparison functions */
```

687

图 B-6 (续)

```
    result[i] = '\0';
    return (result);
}

string ConvertToUpperCase(string s)
{
    string result;
    int i;

    if (s == NULL) {
        Error("NULL string passed to ConvertToUpperCase");
    }
    result = CreateString(strlen(s));
    for (i = 0; s[i] != '\0'; i++) result[i] = toupper(s[i]);
    result[i] = '\0';
    return (result);
}

/* Section 5 -- Functions for converting numbers to strings */

string IntegerToString(int n)
{
    char buffer[MaxDigits];

    sprintf(buffer, "%d", n);
    return (CopyString(buffer));
}

int StringToInteger(string s)
{
    int result;
    char dummy;

    if (s == NULL) {
        Error("NULL string passed to StringToInteger");
    }
    if (sscanf(s, " %d %c", &result, &dummy) != 1) {
        Error("StringToInteger called on illegal number %s", s);
    }
    return (result);
}

string RealToString(double d)
{
    char buffer[MaxDigits];

    sprintf(buffer, "%G", d);
    return (CopyString(buffer));
}

double StringToReal(string s)
{
    double result;
    char dummy;

    if (s == NULL) Error("NULL string passed to StringToReal");
```

689

690

图 B-6 (续)

```
/*
 * Function: RandomReal
 * Usage: d = RandomReal(low, high);
 * ---------------------------------
 * This function returns a random real number in the half-open
 * interval [low .. high), meaning that the result is always
 * greater than or equal to low but strictly less than high.
 */

double RandomReal(double low, double high);

/*
 * Function: RandomChance
 * Usage: if (RandomChance(p)) . . .
 * ---------------------------------
 * The RandomChance function returns TRUE with the probability
 * indicated by p, which should be a floating-point number between
 * 0 (meaning never) and 1 (meaning always). For example, calling
 * RandomChance(.30) returns TRUE 30 percent of the time.
 */

bool RandomChance(double p);

#endif
```

692

图 B-7 (续)

```
/*
 * File: random.c
 * --------------
 * Implements the random.h interface.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "genlib.h"
#include "random.h"

/*
 * Function: Randomize
 * -------------------
 * This function operates by setting the random number
 * seed to the current time. The srand function is
 * provided by the <stdlib.h> library and requires an
 * integer argument. The time function is provided
 * by <time.h>.
 */

void Randomize(void)
{
    srand((int) time(NULL));
}
```

693

图B-8 random.c

索　引

索引中的页码为英文原书页码，与书中边栏页码一致。

! operator (!运算符), 107
#define, 89, 649
#include, 23, 649
% (in printf) (% (在printf中)), 81
% remainder operator (%余数运算符), 45
%% specification (%%说明), 82
%d format (%d 格式), 82
%e format (%e 格式), 82
%f format (%f 格式), 82
%g format (%g 格式), 82
%s format (%s 格式), 82
& operator (&运算符), 458
&& operator (&&运算符), 107
* operator (*运算符), 43, 458
= operator (=运算符), 62
+ operator (+运算符), 43
++ operator (++运算符), 63, 473
+= operator (+=运算符), 62
- operator (-运算符), 43
-- operator (--运算符), 63, 473
-= operator (-=运算符), 62
-> operator (一运算符), 570
. operator (.运算符), 561
/ operator (/运算符), 43
/= operator (/=运算符), 62
; (statement terminator) (; (语句结束符)), 104
?: operator (?: 运算符), 115～117
|| operator (||运算符), 107
\0, 参见null character
\n, 参见newline

A

abacus (算盘), 2
abs, 145, 664
absolute coordinates (绝对坐标), 226
abstract data type (ADT) (抽象数据类型 (ADT)), 319, 615
abstraction (抽象), 224
abstraction boundary (抽象边界), 262
acronym (取首字母的缩写词), 326
Ada, 3
add10.c, 70
add2.c, 26
add2f.c, 35
addlist.c, 72, 124
AddPoint, 563
address (地址), 383
AdjustPen, 239
ADT, 参见abstract data type
adventure games (冒险游戏), 589
Aesop, 261
Alfonso the Wise, 425
algorithm (算法), 9, 185～213, 445, 628
algorithmic design (算法设计), 10
algorithmic efficiency (算法效率), 189, 628
alignment (对齐), 84
allocated size (分配长度), 391
allocation (分配), 385, 478～482
anagram (由颠倒字母顺序而构成的字), 611
analysis of algorithms (算法分析), 445, 628
Analytical Engine (分析机), 2
ANSI C (ANSI C), 15
ANSI string library (ANSI字符串库), 502
answerT, 579
argument (参数), 25, 139
arithmetic expression (算术表达式), 43
arithmetic overflow (算术溢出), 272
array (数组), 375～414
array element (数组元素), 377
array selection (数组选择), 378
array size (数组长度), 377
ASCII, 309
assignment statement (赋值语句), 28, 41
associativity (结合性), 103
atan, 665
atan2, 665
Atanasoff, John, 3
AtEndOfLine, 361
atomic data type (原子数据类型), 106, 659
automatic allocation (自动分配), 478
automatic type conversion (自动类型转换), 48
ave2f.c, 47

cursor（光标），8
cycle（周期），67，120
cyclic cipher（循环加密），336

D

dangling-else problem（悬空else问题），113
data（数据），25
data structure（数据结构），376
data type（数据类型），33
data-driven（数据驱动），588
database（数据库），570
De Morgan's law（德·摩根定律），109
debugging（调试），13，76～80，190～192，270～272
declaring a variable（声明一个变量），27，40
decomposition（分解），165～176
decrement operator（自减运算符），63，473
default clause（default子句），118
default value（缺省值），359
Dequeue，622
dequeue operation（出队操作），616
dereferencing a pointer（间接引用一个指针），458
desk-checking（桌面检查），189
dicetest.c，279
Dickens，Charles，558
dict.h，598
Difference Engine（差分机），2
digitsum.c，122
Dijkstra Edsger，450
directionT，306
disk（磁盘），8，524
divide-and-conquer（分而治之），631
do statement（do语句），655
dot operator（点运算符），561
double，34，212
DrawArc，233
DrawBox，237
drawbox.c，238
DrawCenteredCircle，237
DrawHouse，243
DrawLine，231
DrawLineTo，239
DrawPolarLine，643
dynamic allocation（动态分配），478～482
dynamic array（动态数组），480
dynamic initialization（动态初始化），357

E

Eckert，J. Presper，3
effective size（有效长度），391
efficiency（效率），189，628～637
Einstein，Albert（阿尔伯特·爱因斯坦），263
element type（元素类型），377
elements.c，545
else clause（else子句），113
embedded assignment（赋值嵌套），102
employeeDB，570
empty string（空字符串），322
encapsulation（封装），576
encoding（编码），304
end-of-file marker（文件结束标志），525
ENIAC，3
Enqueue，620
enqueue operation（入队操作），616
Enter key（Enter键），9
enum，305
enumeration type（枚举类型），303
EOF，529
Error，200
error handling（错误处理），201
escape sequence（转义序列），311，650
Euclid，195
Euclid's algorithm（Euclid算法），195
executable file（可执行文件），11
exp，665
exporting a function（导出一个函数），223
expression（表达式），37～51，652～653
extending an interface（扩展一个接口），264
extension（扩展），11

F

fabs，182，665
fact.c，152
Factorial，147，604
FALSE，105
fclose，528，662
feature enhancement（功能增强），14
fgets，536，663
field（字段），559
field width（字段宽度），84
file（文件），11，523～548
FILE *，526
file transfer protocol（FTP）（文件传输协议（FTP）），670
fill.h，372
filling（填充），371
FindChar，325
findcoin.c，427
FindFirstVowel，352，495～496
FindIntegerInArray，427

IntegerToString, 329
integrated circuit (集成电路), 4
interface (接口), 221～251, 259～290
interface entry (接口项), 275
internal state (内部状态), 349
Internet worm (因特网蠕虫), 507
invert.c, 512
isalnum, 314, 666
isalpha, 314, 666
isdigit, 314, 666
IsEven, 148
IsLegalWord, 350
islower, 314, 666
IsPrime, 187, 192
ispunct, 314, 666
isspace, 314, 666
isupper, 314, 666
iteration (迭代), 67, 105
IthChar, 321

J

Jacquard. Joseph Marie, 3
Juster, Norton, 301

K

Kernighan, Brian, 22
keyword (关键词), 39
Koch snowflake (Koch雪花), 642
Koch, Helge von, 642

L

Lao-tzu (老子), 261
layered abstraction (分层抽象), 317～319, 502
leapyear.c, 112
left alignment (左对齐), 84
left-associative (左结合), 103
Leibniz, Gottfried, 2
letter-substitution cipher (字母替代加密), 336
lexical analysis (词法分析), 348
lexicographic order (词典顺序), 324
library (库), 11, 23, 138～140, 221～251, 274～283, 662～667
liftoff.c, 127
line drawing (画线), 231
line graph (折线图), 420
linear search (线性搜索), 431～432
linear time (线性时间), 630
linked list (链接表), 628
linking (链接), 12
ListEmployees, 565, 574
ListPermutations, 613
local variable (局部变量), 154, 478
log, 665
logarithm (对数), 435, 635, 665
logical and (逻辑与), 107
logical not (逻辑非), 107
logical operator (逻辑运算符), 107
logical or (逻辑或), 107
long, 211
long double, 212
loop (循环), 67, 119～130
loop invariant (循环不变式), 188
loop-and-a-half problem (循环半途退出问题), 123
Lord Acton, 15
Lord Byron, 3
lvalue (左值), 455

M

main program (主程序), 24
main module (主模块), 341
malloc, 478, 664
Mandelbrot, Benoit, 641
math library (数学库), 139
math.h, 139, 664
matrix (矩阵), 410
Mauchly, John, 3
mean (均值), 417
median (中值), 448
member (成员), 559
memory (内存), 7, 382～388, 478～482
memory allocation (内存分配), 478～482
Merge, 634
merge sort (合并排序), 633～637
merging (合并), 632
microprocessor (微处理器), 4
MinF, 145
mnemonic (助记词), 644
mode (模式), 449
modular arithmetic (取模运算), 169
modular development (模块化开发), 339～366
module (模块), 340
Monte Carlo integration (蒙特卡洛积分), 295
MonthName, 147
Morse code (Morse码), 419
MovePen, 231
multidimensional array (多维数组), 410～413
multiple assignment (多重赋值), 102

RandomReal, 284
randtest.c, 267
RAND_MAX, 266
rational number (有理数), 594
read-an-integer idiom (读入一个整型数习语), 61
read-until-sentinel idiom (读入一直到一标志习语), 71
ReadEmployeeDatabase, 573
ReadLine, 538
ReadOneEmployee, 573
RealToString, 329
record (记录), 557～589
record selection (记录选择), 561
recursion (递归), 602～614
recursive leap of faith (递归信任), 610
reductionism (归约论), 32
relational operator (关系运算符), 75, 105
relative coordinates (相对坐标), 226
remainder operator (%) (取余运算符 (%)), 45
remove, 664
rename, 664
repeat-N-times idiom (重复N次习语), 65
repfirst.c, 328
return address (返回地址), 160
Return key (Return键), 9, 28
return statement (return语句), 142, 655
returning a value (返回一个值), 28, 139
reverse.c, 399
Reynolds, Malvina, 375
right alignment (右对齐), 84
right-associative (右结合), 103
right-hand rule (右手规则), 214
ring buffer (环缓冲区), 627
Ritchie, Dennis, 15, 22
root name (根名), 11

S

sample run (运行示例), 9
scalar type (标量类型), 307
scanf, 539－547, 663
scanner.c, 361
scanner.h, 353
scanning (扫描), 348
Schickard, Wilhelm, 2
scientific notation (科学记数法), 38
scope (范围), 355
searching (查找), 426～437
secondary storage (辅助存储器), 8
seed (种子), 282
selection (选择), 378, 561
selection expression (选择表达式), 378
selection sort (选择排序), 438～446
semantics (语义), 188
semicolon (分号), 104
sentinel (标志), 70
series expansion (级数展开), 201～209
Shakespeare, William (威廉·莎士比亚), 259, 491
short, 211
short-circuit evaluation (简化求值), 109
shorthand assignment (复合赋值运算符), 62
signtest.c, 115
simpio.c, 677
simpio.h, 26, 676
simple statement (简单语句), 100, 653
sin, 139, 665
sizeof, 385
Smith, Al, 557
Software (软件), 6
software engineering (软件工程), 14
software maintenance (软件维护), 13
sort.c, 443
sort.h, 439
sorting (排序), 437－446
SortIntegerArray, 441, 634
source file (源文件), 11
special character (特殊字符), 311
sprintf, 539, 663
sqrt, 139, 665
srand, 282, 664
sscanf, 540, 663
stack frame (栈帧), 154
standard deviation (标准差), 417
standard files (标准文件), 528
statement (语句), 25
static, 356
static allocation (静态分配), 478
static initialization (静态初始化), 357, 404～410, 561～562
stderr, 528
stdin, 528
stdio.h, 24, 525, 662
stdlib.h, 145, 266, 478, 664
stdout, 528
Stein, Gertrude, 523
stepwise refinement (逐步精化), 165～176
Stoppard, Tom, 293
strcat, 508, 667
strchr, 510, 667
strcmp, 509, 666